AMIGA™

Hardware Reference Manual

Commodore Business Machines, Inc.

AMIGA HARDWARE REFERENCE MANUAL

Amiga Hardware Reference Manual

Commodore Business Machines, Inc.

Amiga Technical Reference Series

Addison-Wesley Publishing Company, Inc.

Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan

Library of Congress Cataloging-in-Publication Data
Main entry under title:

Amiga hardware reference manual.
Includes index. 1. Amiga (Computer) I. Commodore Business Machines.
QA76.8.A177A65 1986 004.165 85-26650
ISBN 0-201-11077-6

FGHIJ-BA-898

The text of this manual was written by Robert Peck, Susan Deyl, Jay Miner, and Chris Raymond.

Special thanks to Bill Kolb, Dave Needle, Lee Ho, and Dale Luck.

COPYRIGHT © 1986 by Commodore-Amiga, Inc.

This manual is copyrighted and all rights are reserved. This document may not, in whole or in part, be copied, photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without prior consent, in writing, from Commodore-Amiga, Inc.

DISCLAIMERS

COMMODORE-AMIGA, INC., ("COMMODORE") MAKES NO WARRANTIES, EITHER EXPRESSED OR IMPLIED, WITH RESPECT TO THE PROGRAMS DESCRIBED HEREIN, THEIR QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. THESE PROGRAMS ARE SOLD "AS IS." THE ENTIRE RISK AS TO THEIR QUALITY AND PERFORMANCE IS WITH THE BUYER. SHOULD THE PROGRAMS PROVE DEFECTIVE FOLLOWING PURCHASE, THE BUYER (AND NOT THE CREATOR OF THE PROGRAMS, COMMODORE, THEIR DISTRIBUTORS OR THEIR RETAILERS) ASSUMES THE ENTIRE COST OF ALL NECESSARY DAMAGES. IN NO EVENT WILL COMMODORE BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE PROGRAMS EVEN IF IT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME LAWS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES OR LIABILITIES FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY.

THE AMIGA COMPUTER MEETS TITLE 47 OF THE CODE OF FEDERAL REGULATIONS FOR CLASS B COMPUTING EQUIPMENT. THE FCC STRICTLY FORBIDS THE USE OF UNSHIELDED CABLE TO ANY AMIGA CONNECTORS WITH THE EXCEPTION OF AC POWER. COMMODORE-AMIGA SHALL NOT BE HELD LIABLE FOR INTERFERENCE GENERATED BY PERIPHERALS NOT AUTHORIZED IN WRITING BY COMMODORE-AMIGA, INC.

Amiga is a trademark of Commodore-Amiga, Inc.

CBM Product Number 327272-02

Sixth Printing, August 1988

PREFACE

This manual provides information about the Amiga[tm] graphics and audio hardware and about how the Amiga talks to the outside world through peripheral devices. A portion of this manual is a tutorial on writing assembly language programs to directly control the Amiga's graphics and hardware.

This book is intended for the following audiences:

- Assembly language programmers who need a more direct way of interacting with the system than the routines described in the *Amiga ROM Kernel Manual.* You can find information here to help you make your programs run faster or do things that the ROM kernel routines don't do.
- Anyone who wants to add new peripherals to the Amiga or just wants to know how the hardware works.

We suggest that you use this book according to your level of familiarity with the Amiga system. Here are some suggestions:

- If this is your initial exposure to the Amiga, read chapter 1, which gives a survey of all the hardware features and a brief rundown of graphics and audio effects created by hardware interaction.
- If you are already familiar with the system and want to acquaint yourself with how the various bits in the hardware registers govern the way the system functions, browse through chapters 2 through 8. Examples are included in these chapters.
- For advanced users, the appendixes give a concise summary of the entire register set and the uses of the individual bits. Once you are familiar with the effects of changes in the various bits, you may wish to refer more often to the appendixes than to the explanatory chapters.

Here is a brief overview of the contents:

Chapter 1, *Introduction.* An overview of the hardware and survey of the Amiga's graphics and audio features.

Chapter 2, *Coprocessor Hardware.* Using the Copper coprocessor to control the entire graphics and audio system; directing mid-screen modifications in graphics displays and directing register changes during the time between displays.

Chapter 3, *Playfield Hardware.* Creating, displaying and scrolling the playfields, one of the basic display elements of the Amiga; how the Amiga produces multi-color, multi-graphical bit-mapped displays.

Chapter 4, *Sprite Hardware.* Using the eight sprite direct-memory access (DMA) channels to make sprite movable objects; creating their data structures, displaying and moving them, reusing the DMA channels.

Chapter 5, *Audio Hardware.* Overview of sampled sound; how to produce quality sound, simple and complex sounds, and modulated sounds.

Chapter 6, *Blitter Hardware.* Using the blitter DMA channel to create animation effects and draw lines into playfields.

Chapter 7, *System Control Hardware.* Using the control registers to define depth arrangement of graphics objects, detect collisions between graphics objects, control direct memory access, and control interrupts.

Chapter 8, *Interface Hardware.* How the Amiga talks to the outside world through controller ports, keyboard, audio jacks and video connectors, serial and parallel interfaces; information about the disk controller and RAM expansion slot.

Appendixes. Alphabetical and address-order listings of all the graphics and audio system registers and the functions of their bits, system memory map, descriptions of internal and external connectors, specifications for the peripheral interface ports, and specifications for the keyboard.

Glossary. After the appendixes, there is a glossary of important terms.

You may wish to look at the following books and manuals for further information about the Amiga:

- The *Amiga ROM Kernel Manual* contains information about the Exec multitasking routines and is the source for all the C language primitives for Amiga graphics, animation, and audio.

- The following manuals contain information about the AmigaDOS operating system:

    - *AmigaDOS User's Manual*

- *AmigaDOS Developer's Manual*

- *AmigaDOS Technical Reference Manual*

It is our policy to make certain that the information contained here is accurate, consistent, and up to date. If you should find any material confusing, inaccurate, or incomplete, please feel free to contact Amiga with your questions or comments.

Table of Contents

Chapter 1 INTRODUCTION 1
Components of the Amiga 1
THE MC 68000 AND THE AMIGA SPECIAL-PURPOSE HARDWARE 2
VCR AND DIRECT CAMERA INTERFACE 5
PRIMARY AND SECONDARY MEMORY 5
PERIPHERALS 5
System Expandability and Adaptability 6

Chapter 2 COPROCESSOR HARDWARE 7
Introduction 7
ABOUT THIS CHAPTER 8
What is a Copper Instruction? 8
The MOVE Instruction 9
The WAIT Instruction 11
HORIZONTAL BEAM POSITION 12
VERTICAL BEAM POSITION 12
THE COMPARISON ENABLE BITS 13
Using the Copper Registers 13
LOCATION REGISTERS 13
JUMP STROBE ADDRESS 14
CONTROL REGISTER 14
Putting Together a Copper Instruction List 15
COMPLETE SAMPLE COPPER LIST 17
LOOPS AND BRANCHES 19
Starting and Stopping the Copper 19
STARTING THE COPPER AFTER RESET 19
STOPPING THE COPPER 19
Advanced Topics 20
THE SKIP INSTRUCTION 20
COPPER LOOPS AND BRANCHES AND COMPARISON ENABLE 21
USING THE COPPER IN INTERLACED MODE 22
USING THE COPPER WITH THE BLITTER 23
THE COPPER AND THE 68000 24
Summary of Copper Instructions 24

Chapter 3 PLAYFIELD HARDWARE 27
- Introduction 27
  - ABOUT THIS CHAPTER 28
  - PLAYFIELD FEATURES 28
- Forming a Basic Playfield 33
  - HEIGHT AND WIDTH OF THE PLAYFIELD 34
  - BIT-PLANES AND COLOR 34
  - SELECTING RESOLUTION 38
  - ALLOCATING MEMORY FOR BIT-PLANES 41
  - CODING THE BIT-PLANES FOR CORRECT COLORING 44
  - DEFINING THE SIZE OF THE DISPLAY WINDOW 46
  - TELLING THE SYSTEM HOW TO FETCH AND DISPLAY DATA 49
  - DISPLAYING AND REDISPLAYING THE PLAYFIELD 52
  - ENABLING THE COLOR DISPLAY 52
  - SUMMARY 53
  - EXAMPLES OF FORMING BASIC PLAYFIELDS 55
- Forming a Dual-playfield Display 58
- Bit-Plane Assignment in Dual-playfield Mode 60
  - COLOR REGISTERS IN DUAL-PLAYFIELD MODE 62
  - DUAL-PLAYFIELD PRIORITY AND CONTROL 64
  - ACTIVATING DUAL-PLAYFIELD MODE 64
  - SUMMARY 65
- Bit-planes and Display Windows of All Sizes 65
  - WHEN THE BIG PICTURE IS LARGER THAN THE DISPLAY WINDOW 65
  - MAXIMUM DISPLAY WINDOW SIZE 72
- Moving (Scrolling) Playfields 73
  - VERTICAL SCROLLING 73
  - HORIZONTAL SCROLLING 74
  - SUMMARY 78
- Advanced Topics 79
  - INTERACTIONS— PLAYFIELDS AND OTHER OBJECTS 79
  - HOLD-AND-MODIFY MODE 79
  - FORMING A DISPLAY WITH SEVERAL DIFFERENT PLAYFIELDS 82
  - USING AN EXTERNAL VIDEO SOURCE 82
  - SUMMARY OF PLAYFIELD REGISTERS 83
- Summary of Color Selection 86
  - COLOR REGISTER CONTENTS 86
  - SOME SAMPLE COLOR REGISTER CONTENTS 86
  - COLOR SELECTION IN LOW-RESOLUTION MODE 87
  - COLOR SELECTION IN HOLD-AND-MODIFY MODE 89
  - COLOR SELECTION IN HIGH-RESOLUTION MODE 89

Chapter 4 SPRITE HARDWARE 91
- Introduction 91
  - ABOUT THIS CHAPTER 92
- Forming a Sprite 92
  - SCREEN POSITION 92
  - SIZE OF SPRITES 95
  - SHAPE OF SPRITES 95
  - SPRITE COLOR 96
  - DESIGNING A SPRITE 98
  - BUILDING THE DATA STRUCTURE 99
- Displaying a Sprite 104
  - SELECTING A DMA CHANNEL AND SETTING THE POINTERS 105
  - RESETTING THE ADDRESS POINTERS 106
  - SPRITE DISPLAY EXAMPLE 106
- Moving a Sprite 108
- Creating Additional Sprites 110
- Reusing Sprite DMA Channels 111
- Overlapped Sprites 114
- Attached Sprites 116
- Manual Mode 119
- Sprite Hardware Details 120
- Summary of Sprite Registers 124
  - POINTERS 124
  - CONTROL REGISTERS 125
  - DATA REGISTERS 126
- Summary of Sprite Color Registers 127

Chapter 5 AUDIO HARDWARE 131
- Introduction 131
  - INTRODUCING SOUND GENERATION 132
  - THE AMIGA SOUND HARDWARE 135
- Forming and Playing a Sound 136
  - DECIDING WHICH CHANNEL TO USE 136
  - CREATING THE WAVEFORM DATA 137
  - TELLING THE SYSTEM ABOUT THE DATA 138
  - SELECTING THE VOLUME 139
  - SELECTING THE DATA OUTPUT RATE 140
  - PLAYING THE WAVEFORM 143
  - STOPPING THE AUDIO DMA 144
  - SUMMARY 145
  - EXAMPLE 145
- Producing Complex Sounds 147
  - JOINING TONES 147

PLAYING MULTIPLE TONES AT THE SAME TIME 149
MODULATING SOUND 149
Producing High-quality Sound 152
MAKING WAVEFORM TRANSITIONS 152
SAMPLING RATE 152
EFFICIENCY 153
NOISE REDUCTION 154
ALIASING DISTORTION 154
LOW-PASS FILTER 156
Using Direct (Non-DMA) Audio Output 157
The Equal-tempered Musical Scale 158
Decibel Values for Volume Ranges 160
The Audio State Machine 161

Chapter 6 BLITTER HARDWARE 165
Introduction 165
Data Copying 167
Pointers and Modulos 168
Ascending and Descending Addressing 170
Rectangular or Linear Address Scanning 171
Blitter Logic Operations 171
DESIGNING THE LF CONTROL BYTE WITH LOGIC EQUATIONS 172
DESIGNING THE LF CONTROL BYTE WITH VENN DIAGRAMS 175
Shifting 177
Masking 178
Zero Detection 179
Area Filling 180
INCLUSIVE (NORMAL) AREA FILLING 180
EXCLUSIVE AREA FILLING 182
Line Drawing 183
OCTANTS IN LINE DRAWING 184
Blitter Operations and System DMA 186
BLITTER DMA PRIORITY 186
DMA TIME SLOT ALLOCATION 186
BIT-PLANE/PROCESSOR BUS SHARING 189
EFFECTS OF DIFFERENT DISPLAY SIZES 191
EFFECTS OF BLITTER OPERATION 191
Complete Blitter Example 193
Blitter Block Diagram 195

Chapter 7 SYSTEM CONTROL HARDWARE **197**
- Introduction 197
- Video Priorities 198
  - FIXED SPRITE PRIORITIES 198
  - HOW SPRITES ARE GROUPED 199
  - UNDERSTANDING VIDEO PRIORITIES 199
  - SETTING THE PRIORITY CONTROL REGISTER 200
- Collision Detection 202
  - HOW COLLISIONS ARE DETERMINED 202
  - HOW TO INTERPRET THE COLLISION DATA 202
  - HOW COLLISION DETECTION IS CONTROLLED 203
- Beam Position Detection 205
  - USING THE BEAM POSITION COUNTER 205
- Interrupts 207
  - NONMASKABLE INTERRUPT 207
  - MASKABLE INTERRUPTS 207
  - USER INTERFACE TO THE INTERRUPT SYSTEM 207
  - INTERRUPT CONTROL REGISTERS 208
  - SETTING AND CLEARING BITS 208
- DMA Control 212

Chapter 8 INTERFACE HARDWARE **215**
- Introduction 215
- Controller Port Interface 216
  - READING THE CONTROLLER PORT 217
- Disk Controller 227
  - DISK SELECTION, CONTROL, AND SENSING 228
  - OTHER REGISTERS IN DISK OPERATIONS 232
  - DISK INTERRUPTS 235
- The Keyboard 236
  - HOW THE KEYBOARD DATA IS RECEIVED 236
  - TYPE OF DATA RECEIVED 236
  - LIMITATIONS OF THE KEYBOARD 239
- Parallel Input/Output Interface 240
- Serial Interface 240
  - INTRODUCTION TO SERIAL CIRCUITRY 240
  - SETTING THE BAUD RATE 240
  - SETTING THE RECEIVE MODE 241
  - CONTENTS OF THE RECEIVE DATA REGISTER 241
  - HOW OUTPUT DATA IS TRANSMITTED 243
  - SPECIFYING THE REGISTER CONTENTS 244
- Audio Output Connections 245
- Display Output Connections 245

Appendix A Register Summary—Alphabetical Order A-1

Appendix B Register Summary—Address Order B-1

Appendix C Custom Chip Pin Allocation List C-1

Appendix D System Memory Map D-1

Appendix E Interfaces E-1

Appendix F Peripheral Interface Adapters F-1

Appendix G Amiga Auto-configuration Architecture G-1

Appendix H Keyboard H-1

Glossary Glossary-1

Index Index-1

Figures

Figure 2-1 Interlaced Bit-Plane in RAM - 400 Lines Long 23
Figure 3-1 How the Video Display Picture Is Produced 29
Figure 3-2 What Is a Pixel? 30
Figure 3-3 How Bit-planes Select a Color 32
Figure 3-4 Significance of Bit-Plane Data in Selecting Colors 33
Figure 3-5 Interlacing 39
Figure 3-6 Effect of Interlaced Mode on Edges of Objects 40
Figure 3-7 Memory Organization for a Basic Bit-Plane 43
Figure 3-8 Combining Bit-planes 45
Figure 3-9 Positioning the On-screen Display 47
Figure 3-10 Data Fetched for the First Line When Modulo = 0 50
Figure 3-11 Data Fetched for the Second Line When Modulo = 0 51
Figure 3-12 A Dual-playfield Display 59
Figure 3-13 How Bit-Planes Are Assigned to Dual Playfields 61
Figure 3-14 Memory Picture Larger than the Display 66
Figure 3-15 Data Fetch for the First Line When Modulo = 40 67
Figure 3-16 Data Fetch for the Second Line When Modulo = 40 67
Figure 3-17 Data Layout for First Line—Right Half of Big Picture 68
Figure 3-18 Data Layout for Second Line—Right Half of Big Picture 68
Figure 3-19 Display Window Horizontal Starting Position 70
Figure 3-20 Display Window Vertical Starting Position 70
Figure 3-21 Display Window Horizontal Stopping Position 71
Figure 3-22 Display Window Vertical Stopping Position 72
Figure 3-23 Vertical Scrolling 74
Figure 3-24 Horizontal Scrolling 75
Figure 3-25 Memory Picture Larger Than the Display Window 77
Figure 3-26 Data for Line 1 - Horizontal Scrolling 77
Figure 3-27 Data for Line 2 - Horizontal Scrolling 77
Figure 4-1 Defining Sprite On-screen Position 93
Figure 4-2 Position of Sprites 94
Figure 4-3 Shape of Spaceship 96
Figure 4-4 Sprite with Spaceship Shape Defined 96
Figure 4-5 Sprite Color Definition 97
Figure 4-6 Color Register Assignments 98
Figure 4-7 Data Structure Layout 101

Figure 4-8 Sprite Priority 111
Figure 4-9 Typical Example of Sprite Reuse 112
Figure 4-10 Typical Data Structure for Sprite Re-use 113
Figure 4-11 Overlapping Sprites (Not Attached) 115
Figure 4-12 Placing Sprites Next to Each Other 116
Figure 4-13 Sprite Control Circuitry 122
Figure 5-1 Sine Waveform 133
Figure 5-2 Digitized Amplitude Values 135
Figure 5-3 Example Sine Wave 142
Figure 5-4 Waveform with Multiple Cycles 153
Figure 5-5 Frequency Domain Plot of Low-Pass Filter 155
Figure 5-6 Noise-free Output (No Aliasing Distortion) 155
Figure 5-7 Some Aliasing Distortion 156
Figure 5-8 Audio State Diagram 164
Figure 6-1 How Images are Stored in Memory 168
Figure 6-2 Bit-plane Image Larger than the Blitter Source Window 169
Figure 6-3 Blitter Minterm Venn Diagram 176
Figure 6-4 A Packed Font 178
Figure 6-5 Blitter Masking Example 179
Figure 6-6 Area-fill Example — Bar Chart 180
Figure 6-7 Use of the FCI Bit - Bit Is a 0 181
Figure 6-8 Use of the FCI Bit - Bit Is a 1 182
Figure 6-9 Single-Point Vertex Example 183
Figure 6-10 Octants for Line Drawing 185
Figure 6-11 DMA Time Slot Allocation 188
Figure 6-12 Normal 68000 Cycle 189
Figure 6-13 Time Slots Used by a Six-bit-plane Display 190
Figure 6-14 Time Slots Used by a High-resolution Display 190
Figure 7-1 Inter-Sprite Fixed Priorities 198
Figure 7-2 Analogy for Video Priority 199
Figure 1-1 Controller Plug and Computer Connector 216
Figure 1-2 Typical Paddle Controller Connection 221
Figure 1-3 The Amiga Keyboard, Showing Keycodes in Hexadecimal 239
Figure 1-4 Starting Appearance of SERDAT and Shift Register 244
Figure 1-5 Ending Appearance of Shift Register 244

Chapter 1

INTRODUCTION

The Amiga is a low-cost, high-performance computer with advanced graphics and sound features. This chapter describes the Amiga's hardware components and gives a brief overview of its graphics and sound features.

Components of the Amiga

These are the hardware components of the Amiga:

- Motorola MC 68000 16/32-bit main processor.
- 256K bytes of internal RAM, expandable to 512K.
- 256K bytes of ROM containing a real-time, multi-tasking operating system with sound, graphics, and animation support routines.
- Built-in 3 1/2-inch double-sided disk drive.
- Expansion disk port for connecting up to three additional disk drives, which may be either 3-1/2 inch or 5-1/4 inch, double-sided.
- Fully programmable serial port.
- Fully programmable parallel port.
- Two-button opto-mechanical mouse.
- Two reconfigurable controller ports (for mice, joysticks, paddles, or custom controllers).
- Detached 89-key keyboard with calculator pad, function keys, and cursor keys.
- Ports for simultaneous composite video and analog or digital RGB output.
- Ports for audio output to left and right stereo channels from four special-purpose audio channels.
- Expansion connector that allows you to add RAM, additional disk drives (floppy or hard), peripherals, or coprocessors.

THE MC 68000 AND THE AMIGA SPECIAL-PURPOSE HARDWARE

The Motorola 68000 is a 16/32-bit microprocessor operating at 7.16 megahertz. In the Amiga, the 68000 can address over 8 megabytes of contiguous random access memory (RAM).

The performance of the 68000 is enhanced by a system design that gives it every alternate bus cycle, allowing it to run at full rated speed most of the time. As described in the section below, the special-purpose hardware can steal time from the 68000 for jobs it can do more efficiently than the 68000. Even then, such cycle stealing only blocks the 68000's access to the shared memory. When using ROM or external memory, the 68000 always runs at full speed.

Among other functions, the special-purpose hardware provides the following:

- Bit-plane-generated high-resolution graphics typically producing 320 by 200 non-interlaced displays and 320 by 400 interlaced displays in 32 colors, and 640 by 200 non-interlaced or 640 by 400 interlaced displays in 16 colors. There is also a special mode that allows you to have up to 4,096 colors on-screen simultaneously.

- A custom display coprocessor that allows changes to most of the special-purpose registers in synchronization with the position of the video beam. This allows such special effects as mid-screen changes to the color palette, splitting the screen into multiple horizontal slices, each having different video resolutions and color depths, beam-synchronized interrupt generation for the 68000, and more. The coprocessor can trigger many times per screen. It can trigger in the middle of lines, as well as at the beginning or during the blanking interval. The coprocessor itself can directly affect most of the registers of the special-purpose hardware, freeing the 68000 for general-purpose computing tasks.

- 32 system color registers, each of which contains a twelve-bit number as four bits of RED, four bits of GREEN, and four bits of BLUE intensity information. This allows a system color palette of 4,096 different choices of color for each register. Although an RGB monitor provides the best available output for the system graphics, text, and color, the composite video signal has been carefully designed to provide maximum NTSC compatibility. This signal may be video-taped or fed to a standard composite video monitor.

- Eight reusable 16-bit-wide sprites with up to 15 color choices per sprite pixel (when sprites are paired). A sprite is an easily movable graphics object whose display is entirely independent of the background (called a playfield); sprites can be displayed "over" or "under" this background. A sprite is 16 low-resolution pixels wide and an arbitrary number of lines tall. After producing the last line of a sprite on the screen, a sprite DMA (direct memory access) channel may be used to produce yet another sprite image elsewhere on-screen (with at least one horizontal line between each reuse of a sprite processor). Thus, you can produce many small sprites by simply reusing the sprite processors appropriately.

- Dynamically-controllable inter-object priority, with collision detection. This means that the system can dynamically control the video priority between the sprite objects and the bit-plane backgrounds (playfields). You can control which object or objects appear "on top" at any time.

  Additionally, you can use system hardware to detect collisions between objects and have your program react to such collisions.

- Custom bit-blitter used for high speed data movement, adaptable to bit-plane animation. The blitter has been designed to efficiently retrieve data from up to three sources, combine the data in one of 256 different possible ways, and optionally store the combined data in a destination area. This is one of the situations where the 68000 gives up memory cycles to a DMA channel that can do the job more efficiently. The bit-blitter, in a special mode, draws patterned lines into rectangularly organized memory regions at a speed of about 1 million dots per second; and it can efficiently handle area fill.

- Audio consisting of four low-noise digital channels with independently programmable volume and sampling rate. The audio channels retrieve their control and data via direct memory access. Once started, each channel can automatically play a specified waveform without further processor interaction. Two channels are directed into each of the two stereo audio outputs. The audio channels may be linked together if desired to provide amplitude or frequency modulation or both forms of modulation simultaneously.

- DMA-controlled floppy disk read and write on a full-track basis. This means that the built-in disk can read something over 5.6K bytes of data in a single disk revolution (11 sectors of 512 bytes each).

All of the special functions described above are produced by three custom-designed VLSI circuits, which work in concert with the 68000. These circuits and the 68000 use the shared memory on a fully interleaved basis. Since the 68000 only needs to access the memory bus during each alternate clock cycle in order to run full-speed, the rest of the time the memory bus is free for other activities.

The special-purpose hardware uses the memory bus during these free cycles, effectively allowing the 68000 to run at full rated speed most of the time. We say "most of the time" because there are some occasions when the special-purpose hardware steals memory cycles from the 68000, but with good reason. Specifically, the coprocessor and the data-moving DMA channel called the blitter can each steal time from the 68000 for jobs they can do better than the 68000. Thus, the system DMA channels are designed with maximum performance in mind; the job to be done is performed by the most efficient hardware element available. In addition, sprites, audio, and disk DMA also steal cycles when in operation.

Another primary feature of the Amiga hardware is the ability to dynamically control which part of memory is used for the background display, audio, and sprites. The Amiga is not limited to a small, specific area of RAM for a frame buffer. Instead, the system allows display bit-planes, sprite-processor control lists, coprocessor instruction lists, or audio channel control lists to be located anywhere within the lowest 512K of the memory map.

This same region of memory can be accessed by the bit-blitter. This means, for example, that the user can store partial images at scattered areas of memory and use these images for animation effects by rapidly replacing on-screen material while saving and restoring background images. In fact, the Amiga includes firmware support for display definition and control as well as support for animated objects embedded within playfields.

VCR AND DIRECT CAMERA INTERFACE

In addition to the connections for NTSC composite Amiga video and both digital and analog RGB monitors, the system can be expanded to include a VCR or camera interface. This system is capable of synchronizing with an external video source and replacing the system background color with the external image. This allows for the development of fully integrated video images with computer-generated graphics. Laser disk input is accepted in the same manner.

PRIMARY AND SECONDARY MEMORY

Primary memory in the Amiga consists of 256K bytes of ROM and 256K bytes of RAM. A RAM expansion cartridge is available as an option. Secondary memory is provided by a built-in 3 1/2-inch floppy disk drive. Disks are 80-track, double-sided, and formatted as 11 sectors per track, 512 bytes per sector (over 900,000 bytes per disk). A special utility can read and write disk files compatible with the Apple II[tm]. In addition, the disk controller can read and write 320/360K IBM PC[tm] formatted disks. External 3 1/2-inch or 5 1/4-inch disk drives can be added to the system through the expansion connector.

PERIPHERALS

Circuitry for some of the peripherals resides on one of the custom chips; other chips handle various signals not specifically assigned to any of the custom chips, including modem controls, disk status sensing, disk motor and stepping controls, ROM enable, parallel input/output interface, and keyboard interface.

The Amiga includes a standard RS-232-C serial port for external serial input/output devices.

A detached, professional-quality keyboard is included in the base system. You can store the keyboard beneath the system cabinet. For maximum flexibility, both key-down and key-up signals are sent.

For those who prefer incremental cursor control, there are cursor keys on the keyboard. You can attach many other types of controllers through the two controller ports on the side of the base unit. You can use a mouse, joystick, keypad, trackball, or steering wheel controller in either of the controller ports. (A light pen can be attached to port 0.)

System Expandability and Adaptability

You can add peripheral devices to the Amiga's expansion connector, add additional external RAM on the same expansion connector, or upgrade internal RAM to 512K. Additional disk units may be daisy-chained from a connector at the rear of the unit for a total of three extra drives.

The system software is highly adaptable to other host operating systems. The Amiga's graphics support routines are designed to make the user interface as friendly as possible. New peripheral devices are recognized and used by system software through a well-defined, well-documented linking procedure.

Chapter 2

COPROCESSOR HARDWARE

Introduction

The Copper is a general purpose coprocessor that resides in one of the Amiga's custom chips. It retrieves its instructions via direct memory access (DMA). The Copper can control nearly the entire graphics system, freeing the 68000 to execute program logic; it can also directly affect the contents of most of the chip control registers. It is a very powerful tool for directing mid-screen modifications in graphics displays and for directing the register changes that must occur during the vertical blanking periods. Among other things, it can control register updates, reposition sprites, change the color palette, update the audio channels, and control the blitter.

One of the features of the Copper is its ability to WAIT for a specific video beam position, then MOVE data into a system register. During the WAIT period, the Copper examines the contents of the video beam position counter directly. This means that while the Copper is waiting for the beam to reach a specific position, it does not use the memory bus at all. Therefore, the bus is freed for use by the other DMA channels or by the 68000.

When the WAIT condition has been satisfied, the Copper steals memory cycles from either the blitter or the 68000 to move the specified data into the selected special-purpose register.

The Copper is a two-cycle processor that requests the bus only during odd-numbered memory cycles. This prevents collision with audio, disk, refresh, sprites, and most low-resolution display DMA access, all of which use only the even-numbered memory cycles. The Copper, therefore, needs priority over only the 68000 and the blitter (the DMA channel that handles animation, line drawing, and polygon filling).

As with all the other DMA channels in the Amiga system, the Copper can retrieve its instructions only from the lowest 512K bytes of system memory.

ABOUT THIS CHAPTER

In this chapter, you will learn how to use the special Copper instruction set to organize mid-screen register value modifications and pointer register set-up during the vertical blanking interval. The chapter shows how to organize Copper instructions into Copper lists, how to use Copper lists in interlaced mode, and how to use the Copper with the blitter. The Copper is discussed in this chapter in a general fashion. The chapters that deal with playfields, sprites, audio, and the blitter contain more specific suggestions for using the Copper.

What is a Copper Instruction?

As a coprocessor, the Copper adds its own instruction set to the instructions already provided by the 68000. The Copper has only three instructions, but you can do a lot with them:

- WAIT for a specific screen position specified as x and y coordinates.
- MOVE an immediate data value into one of the special-purpose registers.
- SKIP the next instruction if the video beam has already reached a specified screen position.

All Copper instructions consist of two 16-bit words in sequential memory locations. Each time the Copper fetches an instruction, it fetches both words. The MOVE and SKIP instructions require two memory cycles and two instruction words. Because only the odd memory cycles are requested by the Copper, four memory cycle times are required per instruction. The WAIT instruction requires three memory cycles and six memory cycle times; it takes one extra memory cycle to wake up.

Although the Copper can directly affect only machine registers, it can affect the memory by setting up a blitter operation. More information about how to use the Copper in controlling the blitter can be found in the sections called "Control Register" and "Using the Copper with the Blitter."

The WAIT and MOVE instructions are described below. The SKIP instruction is described in the "Advanced Topics" section.

The MOVE Instruction

The MOVE instruction transfers data from RAM to a register destination. The transferred data is contained in the second word of the MOVE instruction; the first word contains the address of the destination register. This procedure is shown in detail in the section called "Summary of Copper Instructions."

FIRST INSTRUCTION WORD (IR1)

Bit 0	Always set to 0.
Bits 8 - 1	Register destination address (DA8-1).
Bits 15 - 9	Not used, but should be set to 0.

SECOND INSTRUCTION WORD (IR2)

Bits 15 - 0	16 bits of data to be transferred (moved) to the register destination.

The Copper can store data into the following registers:

- Any register whose address is $20 or above.[1]
- Any register whose address is between $10 and $20 if the Copper danger bit is a 1. The Copper danger bit is in the Copper's control register, COPCON, which is described in the "Control Register" section.
- The Copper cannot write into any register whose address is lower than $10.

Appendix B contains all the machine register addresses.

The following example MOVE instructions point bit-plane pointer 1 at $21000 and bit-plane pointer 2 at $25000.[2]

```
DC.W    $00E0,$0002     ;Move $0002 to address $0E0 (BPL1PTH)
DC.W    $00E2,$1000     ;Move $1000 to address $0E2 (BPL1PTL)
DC.W    $00E4,$0002     ;Move $0002 to address $0E4 (BPL2PTH)
DC.W    $00E6,$5000     ;Move $5000 to address $0E6 (BPL2PTL)
```

---

[1] Hexadecimal numbers are distinguished from decimal numbers by the $ prefix.

[2] All sample code segments are in assembler language.

The WAIT Instruction

The WAIT instruction causes the Copper to wait until the video beam counters are equal to (or greater than) the coordinates specified in the instruction. While waiting, the Copper is off the bus and not using memory cycles.

The first instruction word contains the vertical and horizontal coordinates of the beam position. The second word contains enable bits that are used to form a "mask" that tells the system which bits of the beam position to use in making the comparison.

FIRST INSTRUCTION WORD (IR1)

Bit 0	Always set to 1.
Bits 15 - 8	Vertical beam position (called VP).
Bits 7 - 1	Horizontal beam position (called HP).

SECOND INSTRUCTION WORD (IR2)

Bit 0	Always set to 0.
Bit 15	The blitter-finished-disable bit. Normally, this bit is a 1. (See the "Advanced Topics" section below.)
Bits 14 - 8	Vertical position compare enable bits (called VE).
Bits 7 - 1	Horizontal position compare enable bits (called HE).

The following example WAIT instruction waits for scan line 150 ($96) with the horizontal position masked off.

```
DC.W    $9601,$FF00         ;Wait for line 150,
                            ;  ignore horizontal counters
```

The following example WAIT instruction waits for scan line 255 and horizontal position 254. This event will never occur, so the Copper stops until the next vertical blanking interval begins.

```
DC.W    $FFFF,$FFFE         ;Wait for line 255,
                            ;  H = 254 (ends Copper list)
```

The following notes apply to both the WAIT instruction and to the SKIP instruction, which is described below in the "Advanced Topics" section.

HORIZONTAL BEAM POSITION

The horizontal beam position has a value of $0 to $E2. The least significant bit is not used in the comparison, so there are 113 positions available for Copper operations. This corresponds to 4 pixels in low resolution and 8 pixels in high resolution. Horizontal blanking falls in the range of $0F to $35. The standard screen (320 pixels wide) has an unused horizontal portion of $04 to $47 (during which only the background color is displayed).

VERTICAL BEAM POSITION

The vertical beam position can be resolved to one line, with a maximum value of 255. There are actually 262 possible vertical positions. Some minor complications can occur if you want something to happen within these last six or seven scan lines. Because there are only eight bits of resolution for vertical beam position (allowing 256 different positions), one of the simplest ways to handle this is shown below.

Instruction	**Explanation**
[... other instructions ...]	
WAIT for position (0,255)	*At this point, the vertical counter appears to wrap to 0 because the comparison works on the least significant bits of the vertical count.*
WAIT for any horizontal position with vertical position 0 through 6, covering the last 6 lines of the scan before vertical blanking occurs.	***Thus the total of 256 + 6 = 262 lines of video beam travel during which Copper instructions can be executed.***

THE COMPARISON ENABLE BITS

Bits 14-1 are normally set to all 1s. The use of the comparison enable bits is described later in the "Advanced Topics" section.

Using the Copper Registers

There are several machine registers and strobe addresses dedicated to the Copper:

- Location registers
- Jump address strobes
- Control register

LOCATION REGISTERS

The Copper has two sets of location registers:

COP1LCH	High 3 bits of first Copper list address.
COP1LCL	Low 16 bits of first Copper list address.
COP2LCH	High 3 bits of second Copper list address.
COP2LCL	Low 16 bits of second Copper list address.

In accessing the hardware directly, you often have to write to a pair of registers that contains the address of some data. The register with the lower address always has a name ending in "H" and contains the most significant data, or high 3 bits of the address. The register with the higher address has a name ending in "L" and contains the least significant data, or low 15 bits of the address. Therefore, you write the 18-bit address by moving one long word to the register whose name ends in "H." This is because when you write long words with the 68000, the most significant word goes in the lower addressed word.

In the case of the Copper location registers, you write the address to COP1LCH. In the following text, for simplicity, these addresses are referred to as COP1LC or COP2LC.

The Copper location registers contain the two indirect jump addresses used by the Copper. The Copper fetches its instructions by using its program counter and increments the program counter after each fetch. When a jump address strobe is written, the corresponding location register is loaded into the Copper program counter. This causes the Copper to jump to a new location, from which its next instruction will be fetched. Instruction fetch continues sequentially until the Copper is interrupted by another jump address strobe.

NOTE

At the start of each vertical blanking interval, COP1LC is automatically used to start the program counter. That is, no matter what the Copper is doing, when the end of vertical blanking occurs, the Copper is automatically forced to restart its operations at the address contained in COP1LC.

JUMP STROBE ADDRESS

When you write to a Copper strobe address, the Copper reloads its program counter from the corresponding location register. The Copper can write its own location registers and strobe addresses to perform programmed jumps. For instance, you might MOVE an indirect address into the COP2LC location register. Then, any MOVE instruction that addresses COPJMP2 strobes this indirect address into the program counter.

There are two jump strobe addresses:

COPJMP1	Restart Copper from address contained in COP1LC.
COPJMP2	Restart Copper from address contained in COP2LC.

CONTROL REGISTER

The Copper can access some special-purpose registers all of the time, some registers only when a special control bit is set to a 1, some registers not at all. The registers that the Copper can always affect are numbered $20 through $FF inclusive. Those it cannot affect at all are numbered $00 to $0F inclusive. (See appendix B for a list of registers in address order.) The Copper control register is within the third, always protected, group. Thus it takes deliberate action on the part of the 68000 to allow the Copper to write into a specific range of the special-purpose registers.

The Copper control register, called COPCON, contains only one bit, bit #1. This bit, called CDANG (for Copper Danger Bit) protects all registers numbered between $10 and $1F inclusive. This range includes the blitter control registers. When CDANG is 0, these registers cannot be written by the Copper. When CDANG is 1, these registers can be written by the Copper. Preventing the Copper from accessing the blitter control registers prevents a "runaway" Copper (caused by a poorly formed instruction list) from accidentally affecting system memory.

NOTE

The CDANG bit is cleared after a reset.

Putting Together a Copper Instruction List

The Copper instruction list contains all the register resetting done during the vertical blanking interval and the register modifications necessary for making mid-screen alterations. As you are planning what will happen during each display field, you may find it easier to think of each aspect of the display as a separate subsystem, such as playfields, sprites, audio, interrupts, and so on. Then you can build a separate list of things that must be done for each subsystem individually at each video beam position.

When you have created all these intermediate lists of things to be done, you must merge them together into a single instruction list to be executed by the Copper once for each display frame. The alternative is to create this all-inclusive list directly, without the intermediate steps.

For example, the bit-plane pointers used in playfield displays and the sprite pointers must be rewritten during the vertical blanking interval so the data will be properly retrieved when the screen display starts again. This can be done with a Copper instruction list that does the following:

```
WAIT until first line of the display
MOVE data to bit-plane pointer 1
MOVE data to bit-plane pointer 2
MOVE data to sprite pointer 1
and so on
```

As another example, the sprite DMA channels that create movable objects can be reused multiple times during the same display field. You can change the size and shape of the reuses of a sprite; however, every multiple reuse normally uses the same set of colors during a full display frame. You can change sprite colors mid-screen with a Copper instruction list that waits until the last line of the first use of the sprite processor and changes the colors before the first line of the next use of the same sprite processor:

```
WAIT for first line of display
MOVE firstcolor1 to COLOR17
MOVE firstcolor2 to COLOR18
MOVE firstcolor3 to COLOR19
WAIT for last line +1 of sprite's first use
MOVE secondcolor1 to COLOR17
MOVE secondcolor2 to COLOR18
MOVE secondcolor3 to COLOR19
and so on
```

As you create Copper instruction lists, note that the final list must be in the same order as that in which the video beam creates the display. The video beam traverses the screen from position (0,0) in the upper left hand corner of the screen to the end of the display (226,263) in the lower right hand corner. The first 0 in (0,0) represents the x position. The second 0 represents the y position. For example, an instruction that does something at position (0,100) should come after an instruction that affects the display at position (0,60).

Note that because of the form of the WAIT instruction, you can sometimes get away with not sorting the list in strict video beam order. The WAIT instruction causes the Copper to wait until the value in the beam counter is equal to *or greater than* the value in the instruction. This means, for example, if you have instructions following each other like this:

```
WAIT for position (64,64)
MOVE data

WAIT for position (60,60)
MOVE data
```

the Copper will perform *both* moves, even though the instructions are out of sequence. The "greater than" specification prevents the Copper from locking up if the beam has already passed the specified position. A side effect is that the second MOVE below will be performed:

```
WAIT for position (60,60)
MOVE data

WAIT for position (60,60)
MOVE data
```

At the time of the second WAIT in this sequence, the beam counters will be greater than the position shown in the instructions. Therefore, the second MOVE will also be performed.

Note also that the above sequence of instructions could just as easily be

```
WAIT for position (60,60)
MOVE data
MOVE data
MOVE data
```

because multiple moves can follow a single WAIT.

COMPLETE SAMPLE COPPER LIST

The following example shows a complete Copper list. This list is for two bit-planes—one at $21000 and one at $25000. At the top of the screen, the color registers are loaded with the following values:

Register	**Color**
COLOR00	white
COLOR01	red
COLOR02	green
COLOR03	blue

At line 150 on the screen, the color registers are reloaded:

Register	**Color**
COLOR00	black
COLOR01	yellow
COLOR02	cyan
COLOR03	magenta

The complete Copper list follows.

```
COPPERLIST:
     DC.W    $00E0,$0002       ;Move $0002 into address $0E0 (BPL1PTH)
     DC.W    $00E2,$1000       ;Move $1000 into address $0E2 (BPL1PTL)
     DC.W    $00E4,$0002       ;Move $0002 into address $0E4 (BPL2PTH)
     DC.W    $00E6,$5000       ;Move $5000 into address $0E6 (BPL2PTL)
;
; Load color registers
;
     DC.W    $0180,$0FFF       ;Move white into address $180 (COLOR00)
     DC.W    $0182,$0F00       ;Move red into address $182 (COLOR01)
     DC.W    $0184,$00F0       ;Move green into address $184 (COLOR02)
     DC.W    $0186,$000F       ;Move blue into address $186 (COLOR03)
;
; Wait for line 150
;
     DC.W    $9601,$FF00       ;Wait for line 150, ignore horiz. position
;
; Reload color registers
;
     DC.W    $0180,$0000       ;Move black into address $0180 (COLOR00)
     DC.W    $0182,$0FF0       ;Move yellow into address $0182 (COLOR01)
     DC.W    $0184,$00FF       ;Move cyan into address $0184 (COLOR02)
     DC.W    $0186,$0F0F       ;Move magenta into address $0186 (COLOR03)
;
; End Copper list by waiting for the impossible
;
     DC.W    $FFFF,$FFFE       ;Wait for line 255, H = 254 (never happens)
```

For more information about color registers, see chapter 3, "Playfield Hardware."

LOOPS AND BRANCHES

Loops and branches in Copper lists are covered in the "Advanced Topics" section below.

Starting and Stopping the Copper

STARTING THE COPPER AFTER RESET

At power-on or reset time, you must initialize one of the Copper location registers (COP1LC or COP2LC) and write to its strobe address before Copper DMA is turned on. This ensures a known start address and known state. Usually, COP1LC is used because this particular register is reused during each vertical blanking time. The following sequence of instructions shows how to initialize a location register. It is assumed that the user has already created the correct Copper instruction list at location "mycoplist."

```
MOVE.L    MYCOPLIST, a0
MOVE.L    A0, COP1LCH          ;Write both COP1LCH and COP1LCL
MOVE.W    COPJMP1, D0          ;Any access to this location
                               ; forces load from COP1LC to
                               ; Copper program counter
MOVE.W    #SETBIT + COPPERDMA, D0
MOVE.W    D0, DMACONW          ;Enable Copper DMA
```

Now, if the contents of COP1LC are not changed, every time vertical blanking occurs the Copper will restart at the same location for each subsequent video screen. This forms a repeatable loop which, if the list is correctly formulated, will cause the displayed screen to be stable.

STOPPING THE COPPER

No stop instruction is provided for the Copper. To ensure that it will stop and do nothing until the screen display ends and the program counter starts again at the top of the instruction list, the last instruction should be to WAIT for an event that cannot occur. A typical instruction is to WAIT for VP = $FF and HP = $FE. An HP of greater than $E2 is not possible. When the screen display ends and vertical blanking starts, the Copper will automatically be pointed to the top of its instruction list, and this final WAIT instruction never finishes.

You can also stop the Copper by disabling its ability to use DMA for retrieving instructions or placing data. The register called DMACON controls all of the DMA channels. Bit 7, COPEN, enables Copper DMA when set to 1.

For information about controlling the DMA, see chapter 7, "System Control Hardware."

Advanced Topics

THE SKIP INSTRUCTION

The SKIP instruction causes the Copper to skip the next instruction if the video beam counters are equal to or greater than the value given in the instruction.

The contents of the SKIP instruction's words are shown below. They are identical to the WAIT instruction, except that bit 0 of the second instruction word is a 1 to identify this as a SKIP instruction.

FIRST INSTRUCTION WORD (IR1)

Bit 0	Always set to 1.
Bits 15 - 8	Vertical position (called VP).
Bits 7 - 1	Horizontal position (called HP).
	Skip if the beam counter is equal to or greater than these combined bits (bits 15 through 1).

SECOND INSTRUCTION WORD (IR2)

Bit 0	Always set to 1.
Bit 15	The blitter-finished-disable bit. (See "Using the Copper with the Blitter" below.)
Bits 14 - 8	Vertical position compare enable bits (called VE).
Bits 7 - 1	Horizontal position compare enable bits (called HE).

The notes about horizontal and vertical beam position found in the discussion of the WAIT instruction apply also to the SKIP instruction.

The following example SKIP instruction skips the instruction following it if VP (vertical beam position) is greater than or equal to 100 ($64).

```
DC.W  $6401,$FF01   ;If VP >= 100, skip next instruction (ignore HP)
```

COPPER LOOPS AND BRANCHES AND COMPARISON ENABLE

You can change the value in the location registers at any time and use this value to construct loops in the instruction list. Before the next vertical blanking time, however, the COP1LC registers *must* be repointed to the beginning of the appropriate Copper list. The value in the COP1LC location registers will be restored to the Copper's program counter at the start of the vertical blanking period.

Bits 14-1 of instruction word 2 in the WAIT and SKIP instructions specify which bits of the horizontal and vertical position are to be used for the beam counter comparison. The position in instruction word 1 and the compare enable bits in instruction word 2 are tested against the actual beam counters before any further action is taken. A position bit in instruction word 1 is used in comparing the positions with the actual beam counters *if and only if* the corresponding enable bit in instruction word 2 is set to 1. If the corresponding enable bit is 0, the comparison is always true. For instance, if you care only about the value in the last four bits of the vertical position, you set only the last four compare enable bits, bits (11-8) in instruction word 2.

As another example, suppose you want to issue an interrupt each time a total of 16 vertical scan lines has occurred. In addition, you want the interrupts only between lines 80 and 160. The Copper instruction sequence below would do this. The enable "masks" are specified with the instructions.

Before the Copper is told to begin this set of instructions, you would use the 68000 to write the address of LOOP to COP2LC.

```
; Copper list to interrupt the 68000 once every 16 scan lines,
; in the range VP = 80 through VP = 160.
;
  DC.W      $5001,$FFFE         ;Wait for VP = $50, HP = 0
  DC.W      $0F01,$0F00         ;Wait for VP = xxxx1111
;
; The following instruction writes to address $09C, the
; interrupt request register.  Writing $8010 sets the Copper
; interrupt bit in the register, which will interrupt the 68000.
;
  DC.W      $009C,$8010         ;Move $8010 to $09C (interrupt 68000)
  DC.W      $A001,$FF01         ;Skip next instruction if VP >= 160
;
; The next MOVE instruction doesn't actually do a move.  It forces
; the Copper to jump to the address in COP2LC.  This must have been
; previously set by either the Copper or the 68000.  If VP >= 160,
; then this instruction will be skipped.
;
  DC.W      $008A,$0000         ;Move 0 to COPJMP2 (COP2LC
                                ;  previously set)
```

USING THE COPPER IN INTERLACED MODE

An interlaced bit-plane display has twice the normal number of vertical lines on the screen. Whereas a normal display has 200 lines, an interlaced display has 400 lines. In interlaced mode, the video beam scans the screen twice from top to bottom, displaying 200 lines at a time. During the first scan, the odd-numbered lines are displayed. During the second scan, the even-numbered lines are displayed and interlaced with the odd-numbered ones. The scanning circuitry thus treats an interlaced display as two display fields, one containing the even-numbered lines and one containing the odd-numbered lines. Figure 2-1 shows how an interlaced display is stored in memory.

Data on the Screen	Data in Memory
Odd field - line 1	Line 1
Even field - line 1	Line 2
Odd field - line 2	Line 3
Even field - line 2	Line 4
.	.
.	.
.	.
Odd field - last line	Line 399
Even field - last line	Line 400

Figure 2-1: Interlaced Bit-Plane in RAM - 400 Lines Long

The system retrieves data for bit-plane displays by using pointers to the starting address of the data in memory. As you can see, the starting address for the even-numbered fields is one line greater than the starting address for the odd-numbered fields. Therefore, the bit-plane pointer must contain a different value for alternate fields of the interlaced display. This means that two separate Copper instruction lists are required.

To get the Copper to execute the correct list, you set an interrupt to the 68000 just after the first line of the display. When the interrupt is executed, you change the contents of the COP1LC location register to point to the second list. Then, during the vertical blanking interval, COP1LC will be automatically reset to point to the original list.

For more information about interlaced displays, see chapter 3, "Playfield Hardware."

USING THE COPPER WITH THE BLITTER

If the Copper is used to start up a sequence of blitter operations, it must wait for the blitter-finished interrupt before starting another blitter operation. Changing blitter registers while the blitter is operating causes unpredictable results. For just this purpose, the WAIT instruction includes an additional control bit, called BFD (for blitter finished disable). Normally, this bit is a 1 and only the beam counter comparisons control the WAIT.

When the BFD bit is a 0, the logic of the Copper WAIT instruction is modified. The Copper will WAIT until the beam counter comparison is true *and* the blitter has finished. The blitter has finished when the blitter-finished flag is set. This bit should be unset with caution. It could possibly prevent some screen displays or prevent objects from being displayed correctly.

For more information about using the blitter, see chapter 6, "Blitter Hardware."

THE COPPER AND THE 68000

On those occasions when the Copper's instructions do not suffice, you can interrupt the 68000 and use its instruction set instead. The 68000 can poll for interrupt flags set in the INTREQ register by various devices. To interrupt the 68000, use the Copper MOVE instruction to store a 1 into the following bits of INTREQ:

Table 2-1: Interrupting the 68000

Bit Number	Name	Function
15	SET/CLR	Set/Clear control bit. Determines if bits written with a 1 get set or cleared.
4	COPEN	Coprocessor interrupting 68000.

See chapter 7, "System Control Hardware," for more information about interrupts.

Summary of Copper Instructions

The table below shows a summary of the bit positions for each of the Copper instructions. See appendix A for a summary of all registers.

Table 2-2: Copper Instruction Summary

	Move		Wait		Skip	
Bit#	IR1	IR2	IR1	IR2	IR1	IR2
15	X	RD15	VP7	BFD	VP7	BFD
14	X	RD14	VP6	VE6	VP6	VE6
13	X	RD13	VP5	VE5	VP5	VE5
12	X	RD12	VP4	VE4	VP4	VE4
11	X	RD11	VP3	VE3	VP3	VE3
10	X	RD10	VP2	VE2	VP2	VE2
09	X	RD09	VP1	VE1	VP1	VE1
08	DA8	RD08	VP0	VE0	VP0	VE0
07	DA7	RD07	HP8	HE8	HP8	HE8
06	DA6	RD06	HP7	HE7	HP7	HE7
05	DA5	RD05	HP6	HE6	HP6	HE6
04	DA4	RD04	HP5	HE5	HP5	HE5
03	DA3	RD03	HP4	HE4	HP4	HE4
02	DA2	RD02	HP3	HE3	HP3	HE3
01	DA1	RD01	HP2	HE2	HP2	HE2
00	0	RD00	1	0	1	1

X = don't care, but should be a 0 for upward compatibility
IR1 = first instruction word
IR2 = second instruction word
DA = destination address
RD = RAM data to be moved to destination register
VP = vertical beam position bit
HP = horizontal beam position bit
VE = enable comparison (mask bit)
HE = enable comparison (mask bit)
BFD = blitter-finished disable

Chapter 3

PLAYFIELD HARDWARE

Introduction

The screen display of the Amiga consists of two basic parts—playfields, which are sometimes called backgrounds, and sprites, which are easily movable graphics objects. This chapter describes how to directly access hardware registers to form playfields.

ABOUT THIS CHAPTER

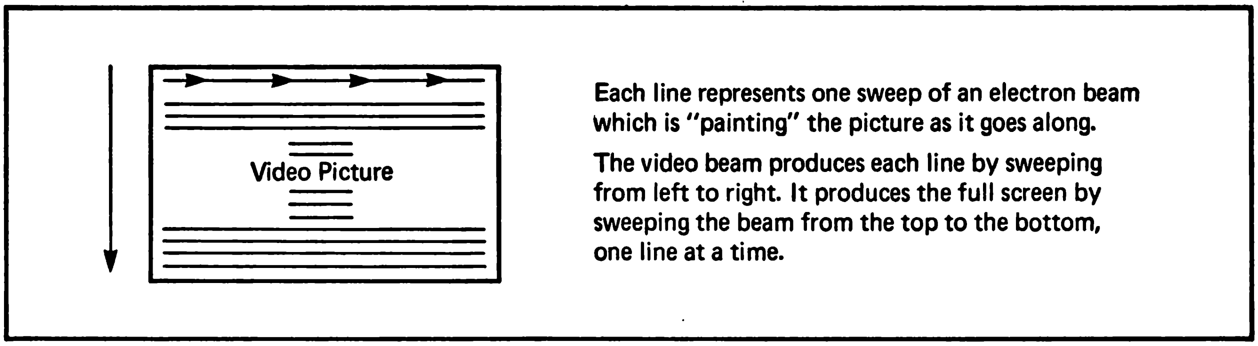
This chapter begins with a brief overview of playfield features, including definitions of some fundamental terms, and continues with the following major topics:

- Forming a single "basic" playfield, which is a playfield the same size as the display screen. This section includes concepts that are fundamental to forming any playfield.
- Forming a dual-playfield display in which one playfield is superimposed upon another. This procedure differs from that of forming a basic playfield in some details.
- Forming playfields of various sizes and displaying only part of a larger playfield.
- Moving playfields by scrolling them vertically and horizontally.
- Advanced topics to help you use playfields in special situations.

For information about movable sprite objects, see chapter 4, "Sprite Hardware." There are also movable playfield objects, which are subsections of a playfield. To move portions of a playfield, you use a technique called playfield animation, which is described in chapter 6, "Blitter Hardware."

PLAYFIELD FEATURES

The Amiga produces its video displays with raster display techniques. You create a graphic display by defining one or more bit-planes in memory and filling them with 1s and 0s to determine the colors in your display. The picture you see on the screen is made up of a series of horizontal video lines displayed one after the other.



Each line represents one sweep of an electron beam which is "painting" the picture as it goes along.

The video beam produces each line by sweeping from left to right. It produces the full screen by sweeping the beam from the top to the bottom, one line at a time.

Figure 3-1: How the Video Display Picture Is Produced

The video beam produces about 262 video lines from top to bottom, of which 200 normally are visible on the screen. Each complete set of 262 lines is called a display field. A complete display field is produced in approximately 1/60th of a second; this is known as the field time. Between display fields, the video beam traverses the lines that are not visible on the screen and returns to the top of the screen to produce another display field.

The display area is defined as a grid of pixels. A pixel is a single picture element, the smallest addressable part of a screen display. The drawings below show what a pixel is and how pixels form displays.

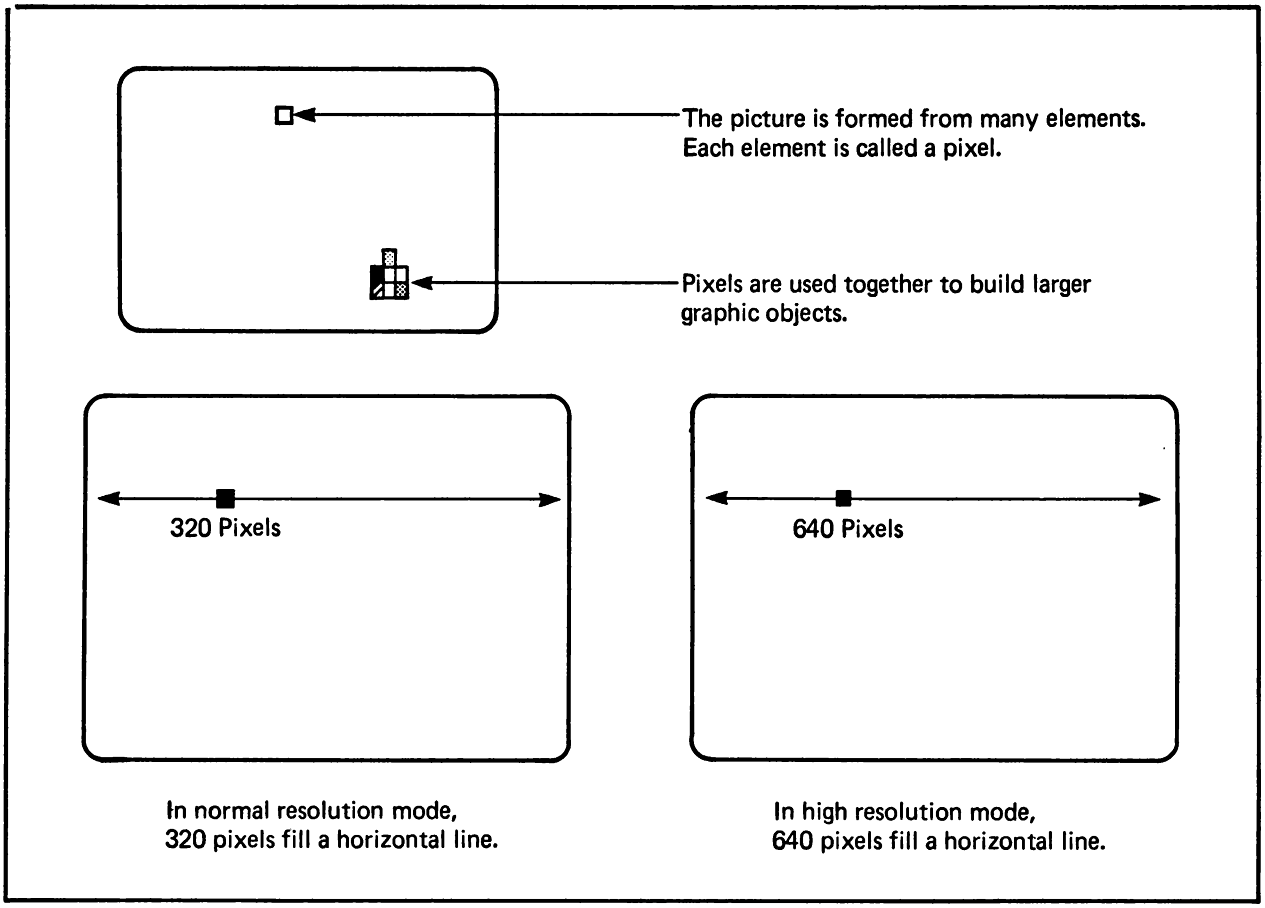

The picture is formed from many elements. Each element is called a pixel.
Pixels are used together to build larger graphic objects.
320 Pixels
640 Pixels
In normal resolution mode, 320 pixels fill a horizontal line.
In high resolution mode, 640 pixels fill a horizontal line.

Figure 3-2: What Is a Pixel?

The Amiga has four basic display modes — interlaced, non-interlaced, low resolution, and high resolution. In non-interlaced mode, the normal playfield has a height of 200 video lines. Interlaced mode gives finer vertical resolution — 400 lines in the same physical display area. In low-resolution mode, the normal playfield has a width of 320 pixels. High-resolution mode gives finer horizontal resolution — 640 pixels in the same physical display area. These modes can be combined, so you can have, for instance, an interlaced, high-resolution display.

Note that the dimensions referred to as "normal" in the previous paragraph are *nominal* dimensions and represent the normal values you should expect to use. Actually, you can display larger playfields; the maximum dimensions are given in the section called "Bit-Planes and Playfields of All Sizes." Also, the dimensions of the playfield in memory are often larger than the playfield displayed on the screen. You choose which part of this larger memory picture to display by specifying a different size for the display window.

A playfield taller than the screen can be scrolled, or moved smoothly, up or down. A playfield wider than the screen can be scrolled horizontally, from left to right or right to left. Scrolling is described in the section called "Moving (Scrolling) Playfields."

In the Amiga graphics system, you can have up to thirty-two different colors in a single playfield, using normal display methods. You can control the color of each individual pixel in the playfield display by setting the bit or bits that control each pixel. A display formed in this way is called a bit-mapped display. For instance, in a two-color display, the color of each pixel is determined by whether a single bit is on or off. If the bit is 0, the pixel is one user-defined color; if the bit is 1, the pixel is another color. For a four-color display, you build two bit-planes in memory. When the playfield is displayed, the two bit-planes are overlapped, which means that each pixel is now two bits deep. You can combine up to five bit-planes in this way. Displays made up of three, four, or five bit-planes allow a choice of eight, sixteen, or thirty-two colors, respectively.

The color of a pixel is always determined by the binary combination of the bits that define it. When the system combines bit-planes for display, the combination of bits formed for each pixel corresponds to the number of a color register. This method of coloring pixels is called *color indirection.* The Amiga has thirty-two color registers, each containing bits defining a user-selected color (from a total of 4,096 possible colors).

Figure 3-3 shows how the combination of up to five bit-planes forms a code that selects which one of the thirty-two registers to use to display the color of a playfield pixel.

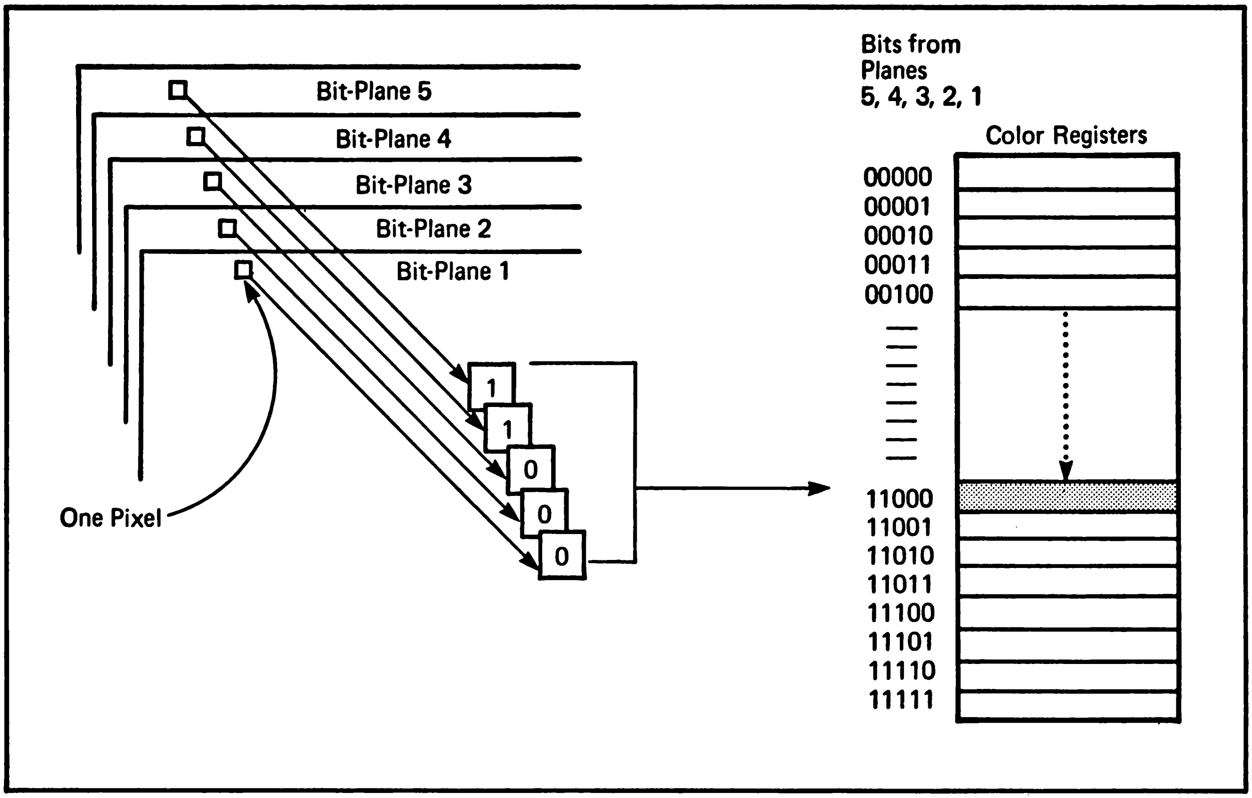

Bits from
Planes
5, 4, 3, 2, 1
Color Registers
Bit-Plane 5
Bit-Plane 4
Bit-Plane 3
Bit-Plane 2
Bit-Plane 1
One Pixel
1
1
0
0
0
00000
00001
00010
00011
00100
11000
11001
11010
11011
11100
11101
11110
11111

Figure 3-3: How Bit-planes Select a Color

Values in the highest numbered bit-plane have the highest significance in the binary number. As shown in figure 3-4, the value in each pixel in the highest-numbered bit-plane forms the leftmost digit of the number. The value in the next highest-numbered bit-plane forms the next bit, and so on.

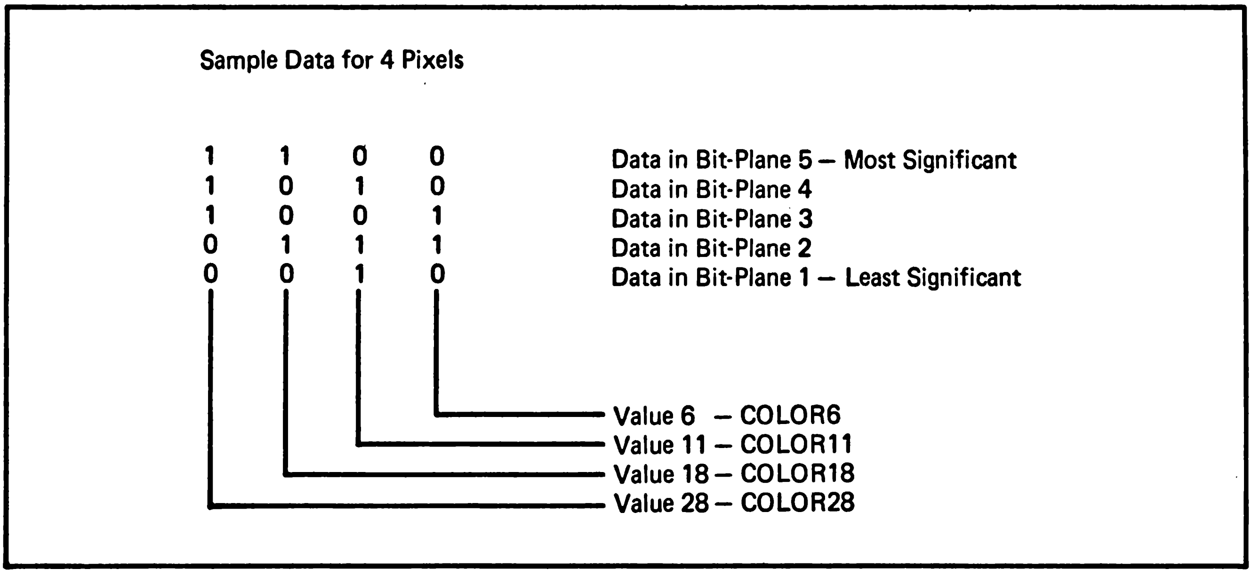

Sample Data for 4 Pixels
1 1 0 0 Data in Bit-Plane 5 – Most Significant
1 0 1 0 Data in Bit-Plane 4
1 0 0 1 Data in Bit-Plane 3
0 1 1 1 Data in Bit-Plane 2
0 0 1 0 Data in Bit-Plane 1 – Least Significant
Value 6 – COLOR6
Value 11 – COLOR11
Value 18 – COLOR18
Value 28 – COLOR28

Figure 3-4: Significance of Bit-Plane Data in Selecting Colors

You also have the choice of defining two separate playfields, each formed from up to three bit-planes. Each of the two playfields uses a separate set of eight different colors. This is called *dual-playfield mode.*

Forming a Basic Playfield

To get you started, this section describes how to directly access hardware registers to form a single basic playfield that is the same size as the video screen. Here, "same size" means that the playfield is the same size as the actual display window. This will leave a small border between the playfield and the edge of the video screen. The playfield usually does not extend all the way to the edge.

To form a playfield, you need to define these characteristics:

- Height and width of the playfield and size of the display window (that is, how much of the playfield actually appears on the screen).
- Color of each pixel in the playfield.

- Horizontal resolution.

- Vertical resolution, or interlacing.

- Data fetch and modulo, which tell the system how much data to put on a horizontal line and how to fetch data from memory to the screen.

In addition, you need to allocate memory to store the playfield, set pointers to tell the system where to find the data in memory, and (optionally) write a Copper routine to handle redisplay of the playfield.

HEIGHT AND WIDTH OF THE PLAYFIELD

To create a playfield that is the same size as the screen, you can use a width of either 320 pixels or 640 pixels, depending upon the resolution you choose. The height is either 200 lines or 400 lines, depending upon whether or not you choose interlaced mode.

BIT-PLANES AND COLOR

You define playfield color by:

1. Deciding how many colors you need and how you want to color each pixel.

2. Loading the colors into the color registers.

3. Allocating memory for the number of bit-planes you need and setting a pointer to each bit-plane.

4. Writing instructions to place a value in each bit in the bit-planes to give you the correct color.

Table 3-1 shows how many bit-planes to use for the color selection you need.

Table 3-1: Colors in a Single Playfield

Number of Colors	Number of Bit-Planes
1 - 2	1
3 - 4	2
5 - 8	3
9 - 16	4
17 - 32	5

The Color Table

The color table contains 32 registers, and you may load a different color into each of the registers. Here is a condensed view of the contents of the color table:

Table 3-2: Portion of the Color Table

Register Name	Contents	Meaning
COLOR0	12 bits	User-defined color for the background area and borders.
COLOR1	12 bits	User-defined color number 1 (For example, the alternate color selection for a two-color playfield).
COLOR2	12 bits	User-defined color number 2.
.	.	
.	.	
.	.	
COLOR31	12 bits	User-defined color number 31.

COLOR0 is always reserved for the background color. The background color shows in any area on the display where there is no other object present and is also displayed outside the defined display window, in the border area.

If you are using the optional genlock board for video input from a camera, VCR, or laser disk, the background color will be replaced by the incoming video display.

Twelve bits of color selection allow you to define, for each of the 32 registers, one of 4,096 possible colors, as shown in table 3-3.

Table 3-3: Contents of the Color Registers

Bits	
Bits 15 - 12	Unused
Bits 11 - 8	Red
Bits 7 - 4	Green
Bits 3 - 0	Blue

Table 3-4 shows some sample color register bit assignments and the resulting colors. At the end of the chapter is a more extensive list.

Table 3-4: Sample Color Register Contents

Contents of the Color Register	**Resulting Color**
$FFF	White
$6FE	Sky blue
$DB9	Tan
$000	Black

Some sample instructions for loading the color registers are shown below:

```
LEA        COLOR0, A0          ;Get address of color register 0 into a0
MOVE.W     #$FFF, (A0)         ;Load white into color register 0
MOVE.W     #$6FE, 2(A0)        ;Load sky blue into color register 1
```

Note that the color registers are write-only. Only by looking at the screen can you find out the contents of each color register. As a standard practice, then, for these and certain other write-only registers, you may wish to keep a "back-up" RAM copy. As you write to the color register itself, you should update this RAM copy. If you do so, you will always know the value each register contains.

Selecting the Number of Bit-Planes

After deciding how many colors you want and how many bit-planes are required to give you those colors, you tell the system how many bit-planes to use.

You select the number of bit-planes by writing the number into the register BPLCON0 (for Bit Plane Control Register 0) The relevant bits are bits 14, 13, and 12, named BPU2, BPU1, and BPU0 (for "Bit Planes Used"). Table 3-5 shows the values to write to these bits and how the system assigns bit-plane numbers.

Table 3-5: Setting the Number of Bit-Planes

Value	**Number of Bit-Planes**	**Name(s) of Bit-Planes**
000	None *	
001	1	PLANE 1
010	2	PLANES 1 and 2
011	3	PLANES 1 - 3
100	4	PLANES 1 - 4
101	5	PLANES 1 - 5
110	6	PLANES 1 - 6 **
111		Value not used.

* Shows only a background color; no playfield is visible.

** Sixth bit-plane is used only in dual-playfield mode and in hold-and-modify mode (described in the section called "Advanced Topics").

NOTE

The bits in the BPLCON0 register are not independently settable. To set any one bit, you must reload them all.

The following example shows how to tell the system to use two low-resolution bit-planes.

```
BPLCON0    EQU      $DFF100            ;BPLCON0 address
           MOVE.W  #$2200,BPLCON0     ;Write to it
```

Because register BPLCON0 is used for setting other characteristics of the display and the bits are not independently settable, the example above also sets other parameters (all of these parameters are described later in the chapter).

- Hold-and-modify mode is turned off.
- Single-playfield mode is set.
- Composite video color is enabled.
- Genlock audio is disabled.
- Light pen is disabled.
- Interlaced mode is disabled.
- External resynchronization is disabled.

SELECTING HORIZONTAL AND VERTICAL RESOLUTION

Standard home television screens are best suited for low-resolution displays. Low-resolution mode provides 320 pixels for each horizontal line. High-resolution monochrome and RGB monitors can produce displays in high-resolution mode, which provides 640 pixels for each horizontal line. If you define an object in low-resolution mode and then display it in high-resolution mode, the object will be only half as wide.

To set horizontal resolution mode, you write to bit 15, HIRES, in register BPLCON0:

High-resolution mode — write 1 to bit 15.
Low-resolution mode — write 0 to bit 15.

Note that in high-resolution mode, you can have up to four bit-planes in the playfield and, therefore, up to 16 colors.

Interlacing allows you to double the number of lines appearing on the video screen. Generally, in non-interlaced mode, 200 lines fill the screen and a playfield of normal size appears full-sized. In interlaced mode, normally, a maximum of 400 lines fill the screen. Twice as much data is displayed in the same vertical area as in non-interlaced mode.

In interlaced mode, the scanning circuitry vertically offsets the start of every other field by half a scan line.

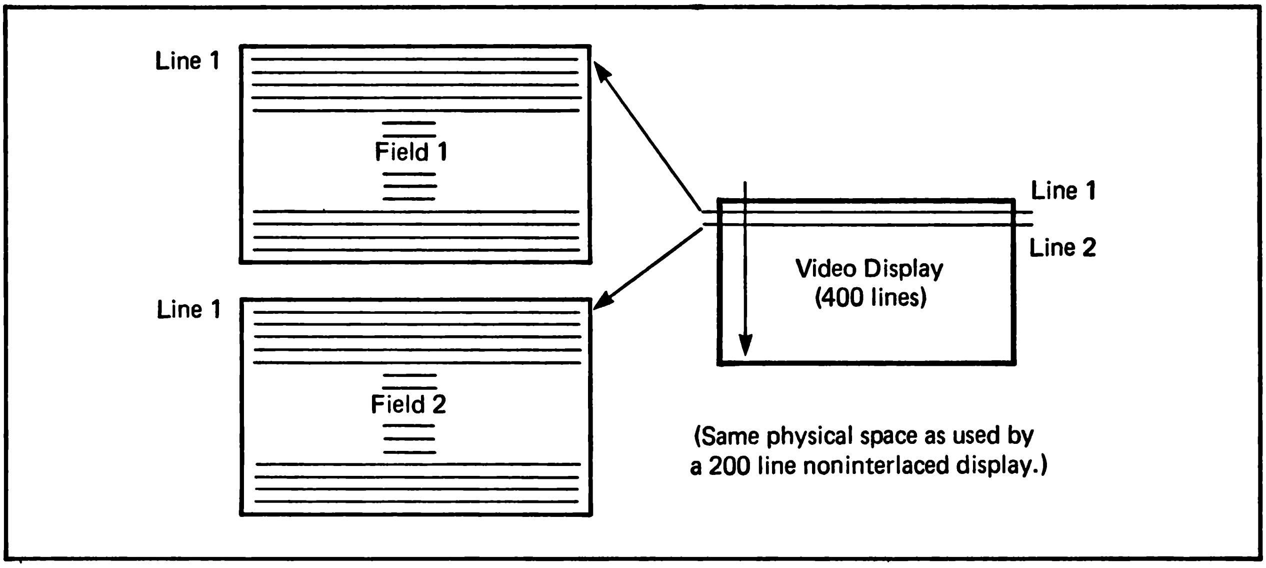

Line 1
Field 1
Line 1
Field 2
Line 1
Line 2
Video Display
(400 lines)
(Same physical space as used by
a 200 line noninterlaced display.)

Figure 3-5: Interlacing

Even though interlaced mode requires a modest amount of extra work in setting registers (as you will see later on in this section), it provides fine tuning that is needed for certain graphics effects. Consider the diagonal line in figure 3-6 as it appears in non-interlaced and interlaced modes. Interlacing eliminates much of the jaggedness or "staircasing" in the edges of the line.

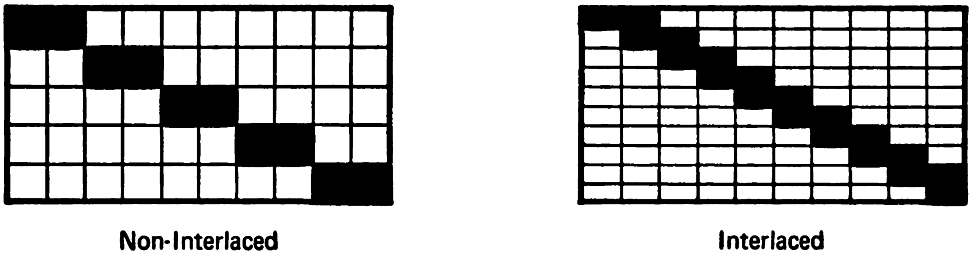


Figure 3-6: Effect of Interlaced Mode on Edges of Objects

When you use the special blitter DMA channel to draw lines or polygons onto an interlaced playfield, the playfield is treated as one display, rather than as odd and even fields. Therefore, you still get the smoother edges provided by interlacing.

To set interlaced or non-interlaced mode, you write to bit 2, LACE, in register BPLCON0:

Interlaced mode — write 1 to bit 2.
Non-interlaced mode — write 0 to bit 2.

As explained above in "Setting the Number of Bit-Planes," bits in BPLCON0 are not independently settable.

The following example shows how to specify high-resolution and interlaced modes.

```
BPLCON0   EQU      $DFF100              ;BPLCON0 address
          MOVE.W   #$A204,BPLCON0       ;Write to it
```

The example above also sets the following parameters that are also controlled through register BPLCON0:

- High-resolution mode is enabled.
- Two bit-planes are used.

- Hold-and-modify mode is disabled.
- Single-playfield mode is enabled.
- Composite video color is enabled.
- Genlock audio is disabled.
- Light pen is disabled.
- Interlaced mode is enabled.
- External resynchronization is disabled.

The amount of memory you need to allocate for each bit-plane depends upon the resolution modes you have selected, because high-resolution or interlaced playfields contain more data and require larger bit-planes.

ALLOCATING MEMORY FOR BIT-PLANES

After you set the number of bit-planes and specify resolution modes, you are ready to allocate memory. A bit-plane consists of an end-to-end sequence of words at consecutive memory locations. To allocate memory, you set the registers that point to the starting memory address of each bit-plane you are using. The starting address is the memory word that contains the bits of the upper left-hand corner of the bit-plane.

Table 3-6 shows how much memory is needed for basic playfields. You may need to balance your color and resolution requirements against the amount of available memory you have.

Table 3-6: Playfield Memory Requirements

Picture Size	**Modes**	**Number of Bytes per Bit-Plane**
320 X 200	Low-resolution, non-interlaced	8,000
320 X 400	Low-resolution, interlaced	16,000
640 X 200	High-resolution, non-interlaced	16,000
640 X 400	High-resolution, interlaced	32,000

A normal low-resolution, non-interlaced display has 320 pixels across each display line and a total of 200 display lines. Each line of the bit-plane for such a display requires 40 bytes (320 bits divided by 8 bits per byte = 40).

A low-resolution, non-interlaced playfield made up of two bit-planes requires 16,000 bytes of memory area. The memory for each bit-plane must be continuous, so you need to have two 8,000-byte blocks of available memory. Figure 3-7 shows an 8,000-byte memory area organized as 200 lines of 40 bytes each, providing 1 bit for each pixel position in the display plane.

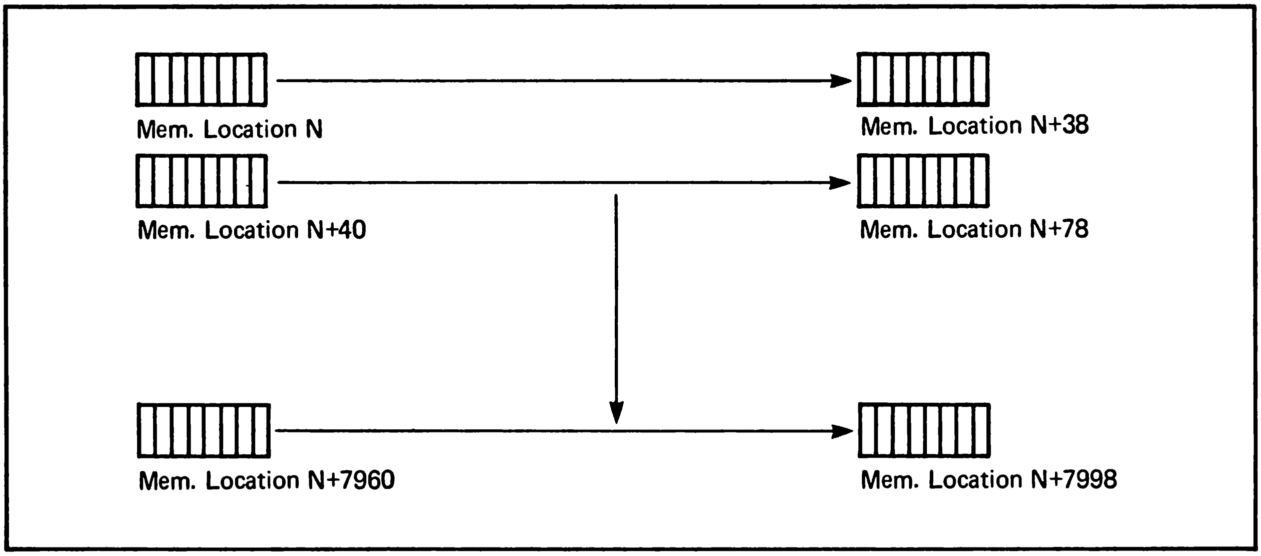


Figure 3-7: Memory Organization for a Basic Bit-Plane

Access to bit-planes in memory is provided by two address registers, BPLxPTH and BPLxPTL, for each bit-plane (12 registers in all). The "x" position in the name holds the bit-plane number; for example BPL1PTH and BPL1PTL hold the starting address of PLANE 1. As usual, pairs of registers with names ending in PTH and PTL contain 19-bit addresses. 68000 programmers may treat these as one 32-bit address and write to them as one long word. You write to the high-order word, which is the register whose name ends in "PTH."

The example below shows how to set the bit-plane pointers. Assuming two bit-planes, one at $21000 and the other at $25000, the processor sets BPL1PT to $21000 and BPL2PT to $25000. Note that this is usually the Copper's task.

```
BPL1PTH EQU     $DFF0E0           ;High three bits of bit-plane 1 pointer
BPL1PTL EQU     $DFF0E2           ;Low fifteen bits
BPL2PTH EQU     $DFF0E4           ;High three bits of bit-plane 2 pointer
BPL2PTL EQU     $DFF0E6           ;Low fifteen bits

        MOVE.L  $21000,BPL1PTH    ;Write bit-plane 1 pointer
        MOVE.L  $25000,BPL2PTH    ;Write bit-plane 2 pointer
```

Note that the memory requirements given here are for the playfield only. You may need to allocate additional memory for other parts of the display — sprites, audio, animation — and for your application programs. Memory allocation for other parts of the display is discussed in the chapters describing those topics.

CODING THE BIT-PLANES FOR CORRECT COLORING

After you have specified the number of bit-planes and set the bit-plane pointers, you can actually write the color register codes into the bit-planes.

A One- or Two-Color Playfield

For a one-color playfield, all you need do is write 0s in all the bits of the single bit-plane as shown in the example below. This code fills a low-resolution bit-plane with the background color (COLOR00) by writing all 0s into its memory area. The bit-plane starts at $21000 and is 8,000 bytes long.

```
        LEA     $21000,A0          ;Point at bit-plane
        MOVE.W  #2000,D0           ;Write 2000 longwords = 8000 bytes
LOOP:   MOVE.L  #0,(A0)+           ;Write out a zero
        SUBQ.W  #1,D0              ;Decrement counter
        BNE     LOOP               ;Loop until bit-plane is filled with 0s
```

For a two-color playfield, you define a bit-plane that has 0s where you want the background color and 1s where you want the color in register 1. The following example code is identical to the last example, except the bit-plane is filled with $FF00FF00 instead of all 0's. This will produce two colors.

```
        LEA     $21000,A0          ;Point at bit-plane
        MOVE.W  #2000,D0           ;Write 2000 longwords = 8000 bytes
LOOP:   MOVE.L  #$FF00FF00,(A0)+   ;Write out $FF00FF00
        SUBQ.W  #1,D0              ;Decrement counter
        BNE     LOOP               ;Loop until bit-plane is full
```

A Playfield of Three or More Colors

For three or more colors, you need more than one bit-plane. The task here is to define each bit-plane in such a way that when they are combined for display, each pixel contains the correct combination of bits. This is a little more complicated than a playfield of one bit-plane. The following examples show a four-color playfield, but the basic idea and procedures are the same for playfields containing up to 32 colors.

Figure 3-8 shows two bit-planes forming a four-color playfield:

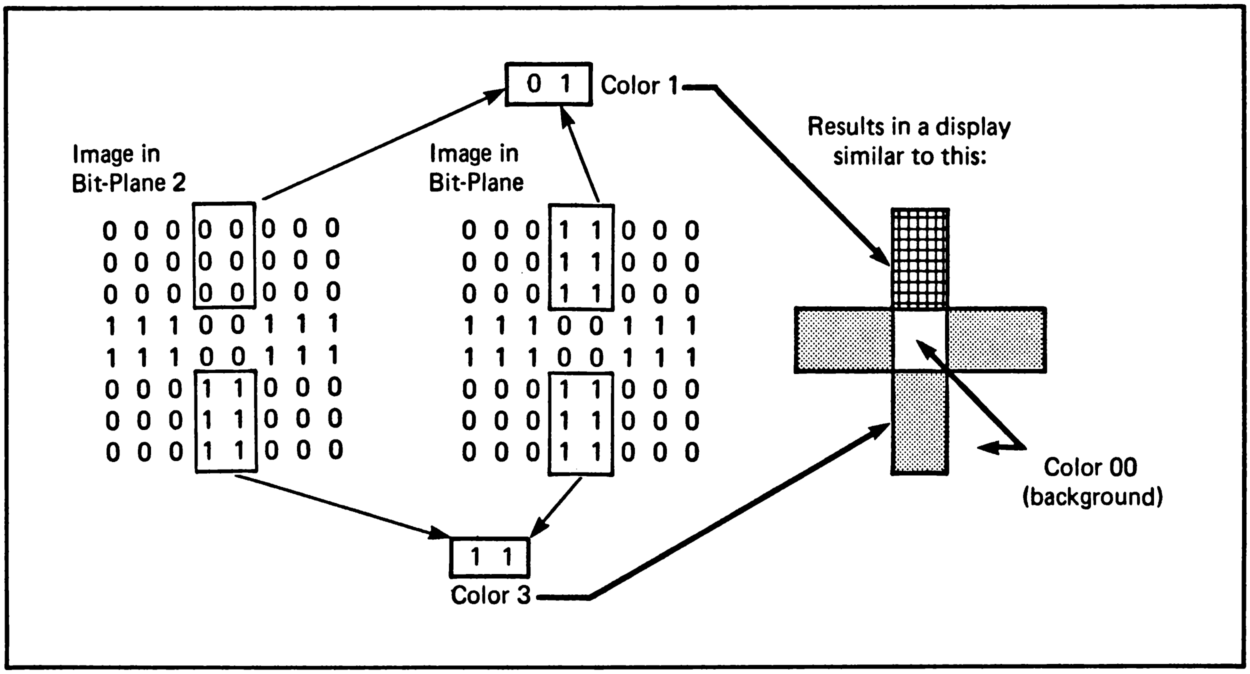

0 1
Color 1
Image in Bit-Plane 2
Image in Bit-Plane
Results in a display similar to this:
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
1 1 1 0 0 1 1 1
1 1 1 0 0 1 1 1
0 0 0 1 1 0 0 0
0 0 0 1 1 0 0 0
0 0 0 1 1 0 0 0
0 0 0 1 1 0 0 0
0 0 0 1 1 0 0 0
0 0 0 1 1 0 0 0
1 1 1 0 0 1 1 1
1 1 1 0 0 1 1 1
0 0 0 1 1 0 0 0
0 0 0 1 1 0 0 0
0 0 0 1 1 0 0 0
Color 00 (background)
1 1
Color 3

Figure 3-8: Combining Bit-planes

You place the correct 1s and 0s in both bit-planes to give each pixel in the picture above the correct color.

In a single playfield you can combine up to five bit-planes in this way. Using five bit-planes allows a choice of 32 different colors for any single pixel. The playfield color selection charts at the end of this chapter summarize the bit combinations for playfields made from four and five bit-planes.

DEFINING THE SIZE OF THE DISPLAY WINDOW

After you have completely defined the playfield, you need to define the size of the display window, which is the actual size of the on-screen display. Adjustment of display window size affects the entire display area, including the border and the sprites, not just the playfield. You cannot display objects outside of the defined display window. Also, the size of the border around the playfield depends on the size of the display window.

The basic playfield described in this section is the same size as the screen display area and also the same size as the display window. This is not always the case; often the display window is smaller than the actual "big picture" of the playfield as defined in memory (the raster). A display window that is smaller than the playfield allows you to display some segment of a large playfield or scroll the playfield through the window. You can also define display windows larger than the basic playfield. These larger playfields and different-sized display windows are described in the section below called "Bit-Planes and Display Windows of All Sizes."

You define the size of the display window by specifying the vertical and horizontal positions at which the window starts and stops and writing these positions to the display window registers. The resolution of vertical start and stop is one scan line. The resolution of horizontal start and stop is one low-resolution pixel. Each position on the screen defines the horizontal and vertical position of some pixel, and this position is specified by the x and y coordinates of the pixel. This document shows the x and y coordinates in this form: (x,y). Although the coordinates begin at (0,0) in the upper left-hand corner of the screen, the first horizontal position normally used is $81 and the first vertical position is $2C. The hardware allows you to specify a starting position before ($81,$2C), but part of the display may not be visible. The difference between the absolute starting position of (0,0) and the normal starting position of ($81,$2C) is the result of the way many video display monitors are designed. To overcome the distortion that can occur at the extreme edges of the screen, the scanning beam sweeps over a larger area than the front face of the screen can display. A starting position of ($81,$2C) centers a normal size display, leaving a border of eight low-resolution pixels around the display window. Figure 3-9 shows the relationship between the normal display window, the visible screen area, and the area actually covered by the scanning beam.

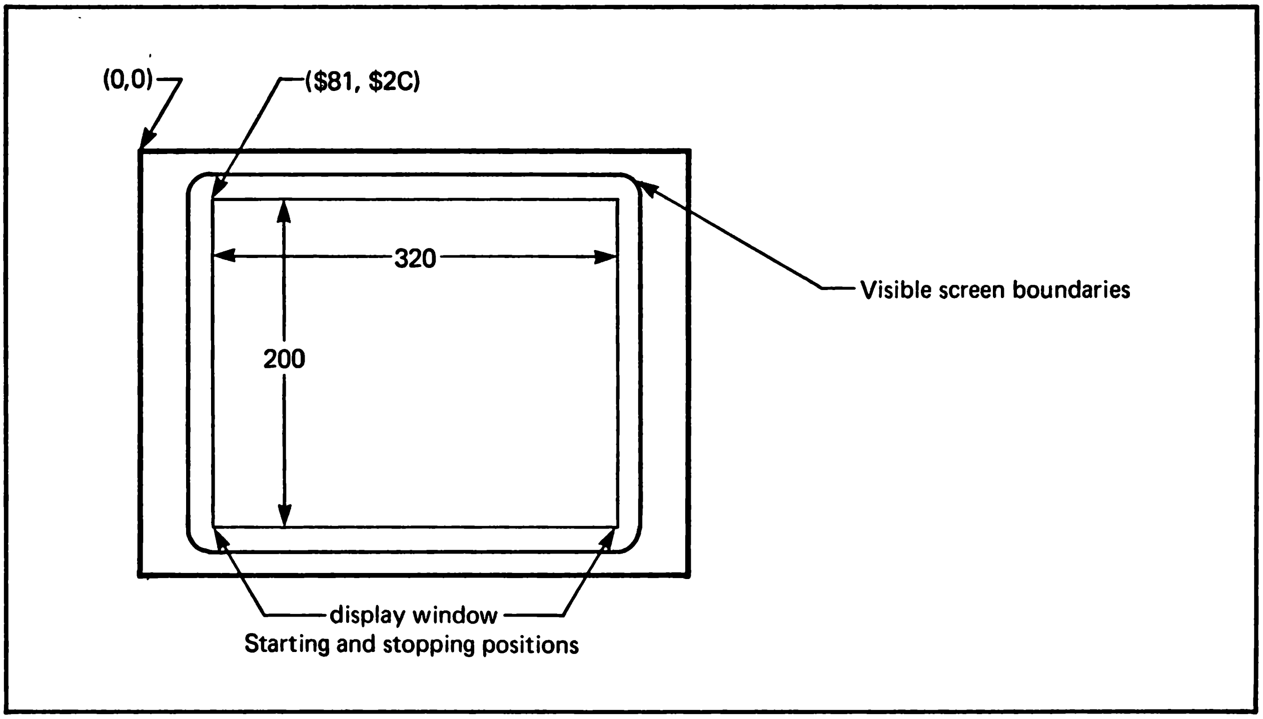


Figure 3-9: Positioning the On-screen Display

Setting the Display Window Starting Position

A horizontal starting position of approximately $81 and a vertical starting position of approximately $2C centers the display on most standard television screens. If you select high-resolution mode (640 pixels horizontally) or interlaced mode (400 lines) the starting position does not change. The starting position is always interpreted in low-resolution, non-interlaced mode. In other words, you select a starting position that represents the correct coordinates in low-resolution, non-interlaced mode.

The register DIWSTRT (for "Display Window Start") controls the display window starting position. This register contains both the horizontal and vertical components of the display window starting positions, known respectively as HSTART and VSTART. The following example sets DIWSTRT for a basic playfield. You write $2C for VSTART and $81 for HSTART.

```
DIWSTRT   EQU      $DFF08E            ;Display window start
                                      ;   register address

          MOVE.W   #$2C81,DIWSTRT     ;Write it out
```

Setting the Display Window Stopping Position

You also need to set the display window stopping position, which is the lower right-hand corner of the display window. If you select high-resolution or interlaced mode, the stopping position does not change. Like the starting position, it is interpreted in low-resolution, non-interlaced mode.

The register DIWSTOP (for Display Window Stop) controls the display window stopping position. This register contains both the horizontal and vertical components of the display window stopping positions, known respectively as HSTOP and VSTOP. The instructions below show how to set HSTOP and VSTOP for the basic playfield, assuming a starting position of ($81,$2C). Note that the HSTOP value you write is the actual value minus 256 ($100). The HSTOP position is restricted to the right-hand side of the screen. The normal HSTOP value is ($1C1) but is written as ($C1).

The VSTOP position is restricted to the lower half of the screen. This is accomplished in the hardware by forcing the MSB of the stop position to be the complement of the next MSB. This allows for a VSTOP position greater than 256 ($100) using only 8 bits. Normally, the VSTOP is set to ($F4).

The normal DIWSTRT is ($2C81). The normal DIWSTOP is ($F4C1).

The following example sets DIWSTOP for a basic playfield to $F4 for the vertical position and $C1 for the horizontal position.

```
DIWSTOP   EQU      $DFF090            ;Display window stop
                                      ;   register address

          MOVE.W   #$F4C1,DIWSTOP     ;Write it out
```

TELLING THE SYSTEM HOW TO FETCH AND DISPLAY DATA

After defining the size and position of the display window, you need to give the system the on-screen location for data fetched from memory. To do this, you describe the horizontal positions where each line starts and stops and write these positions to the data-fetch registers. The data-fetch registers have a four-pixel resolution (unlike the display window registers, which have a one-pixel resolution). Each position specified is four pixels from the last one. Pixel 0 is position 0; pixel 4 is position 1, and so on.

The data-fetch start and display window starting positions interact with each other. It is recommended that data-fetch start values be restricted to a programming resolution of 16 pixels (8 clocks in low-resolution mode, 4 clocks in high-resolution mode). The hardware requires some time after the first data fetch before it can actually display the data. As a result, there is a difference between the value of window start and data-fetch start. In low-resolution mode the difference is 8.5 clocks; in high-resolution mode the difference is 4.5 clocks.

The normal low-resolution DDFSTRT is ($0038). The normal high-resolution DDFSTRT is ($003C). Recall that the hardware resolution of display window start and stop is twice the hardware resolution of data fetch:

($81/2 - 8.5) = ($38)
($81/2 - 4.5) = ($3C)

The relationship between data-fetch start and stop is

DDFSTRT = DDFSTOP - (8*(word count - 1) for low resolution
DDFSTRT = DDFSTOP - (4*(word count - 2) for high resolution

The normal low-resolution DDFSTOP is ($00D0). The normal high-resolution DDFSTOP is ($00D4).

The following example sets data-fetch start to $0038 and data-fetch stop to $00D0 for a basic playfield.

```
DDFSTRT  EQU      $DFF092
DDFSTOP  EQU      $DFF094

         MOVE.W   #$0038,DDFSTRT   ;Write to DDFSTRT
         MOVE.W   #$00D0,DDFSTOP   ;Write to DDFSTOP
```

You also need to tell the system exactly which bytes in memory belong on each horizontal line of the display. To do this, you specify the modulo value. Modulo refers to the number of bytes in memory between the last word on one horizontal line and the beginning of the first word on the next line. Thus, the modulo enables the system to convert bit-plane data stored in linear form (each data byte at a sequentially increasing memory address) into rectangular form (one "line" of sequential data followed by another line). For the basic playfield, where the playfield in memory is the same size as the display window, the modulo is zero because the memory area contains exactly the same number of bytes as you want to display on the screen. Figures 3-10 and 3-11 show the basic bit-plane layout in memory and how to make sure the correct data is retrieved.

The bit-plane address pointers (BPLxPTH and BPLxPTL) are used by the system to fetch the data to the screen. These pointers are dynamic; once the data fetch begins, the pointers are continuously incremented to point to the next *word* to be fetched (data is fetched two bytes at a time). When the end-of-line condition is reached (defined by the data-fetch register, DDFSTOP) the modulo is added to the bit-plane pointers, adjusting the pointer to the first word to be fetched for the next horizontal line.

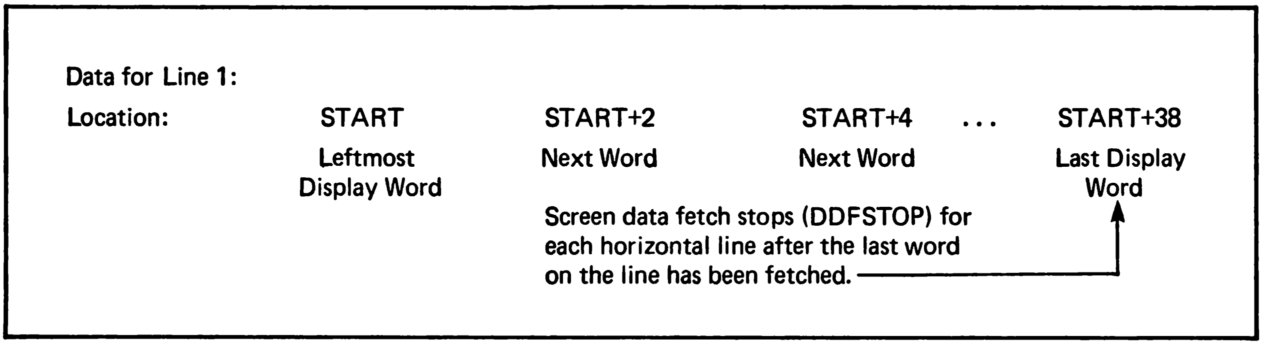

Data for Line 1:
Location:
START
Leftmost Display Word
START+2
Next Word
START+4
Next Word
...
START+38
Last Display Word
Screen data fetch stops (DDFSTOP) for each horizontal line after the last word on the line has been fetched.

Figure 3-10: Data Fetched for the First Line When Modulo = 0

After the first line is fetched, the bit-plane pointers BPLxPTH and BPLxPTL contain the value START+40. The modulo (in this case, 0) is added to the current value of the pointer, so when the pointer begins the data fetch for the next line, it fetches the data you want on that line. The data for the next line begins at memory location START+40.

Data for Line 2:

Location:	START+40	START+42	START+44	...	START+78
	Leftmost Display Word	Next Word	Next Word		Last Display Word

Figure 3-11: Data Fetched for the Second Line When Modulo = 0

Note that the pointers always contain an even number, because data is fetched from the display a *word* at a time.

There are two modulo registers—BPL1MOD for the odd-numbered bit-planes and BPL2MOD for the even-numbered bit-planes.

The following example sets the modulo to 0 for a low-resolution playfield with one bit-plane. The bit-plane is odd-numbered.

```
BPL1MOD  EQU      $DFF108       ;Modulo for odd bit-planes

         MOVE.W   #0,BPL1MOD    ;Set modulo to 0
```

Data Fetch in High-resolution Mode

When you are using high-resolution mode to display the basic playfield, you need to fetch 80 bytes for each line, instead of 40.

Modulo in Interlaced Mode

For interlaced mode, you must redefine the modulo, because interlaced mode uses two separate scannings of the video screen for a single display of the playfield. During the first scanning, the odd-numbered lines are fetched to the screen; and during the second scanning, the even-numbered lines are fetched.

The bit-planes for a full-screen-sized, interlaced display are 400, rather than 200, lines long. Assuming that the playfield in memory is the normal 320 pixels wide, data for the interlaced picture begins at the following locations (these are all byte addresses):

Line 1	START
Line 2	START+40
Line 3	START+80
Line 4	START+120

and so on. Therefore, you use a modulo of 40 to skip the lines in the other field. For odd fields, the bit-plane pointers begin at START. For even fields, the bit-plane pointers begin at START+40.

You can use the Copper to handle resetting of the bit-plane pointers for interlaced displays.

DISPLAYING AND REDISPLAYING THE PLAYFIELD

You start playfield display by making certain that the bit-plane pointers are set and bit-plane DMA is turned on. You turn on bit-plane DMA by writing a 1 to bit BPLEN in the DMACON (for DMA control) register. See chapter 7, "System Control Hardware," for instructions on setting this register.

Each time the playfield is redisplayed, you have to reset the bit-plane pointers. Resetting is necessary because the pointers have been incremented to point to each successive word in memory and must be repointed to the first word for the next display. You write Copper instructions to handle the redisplay or perform this operation as part of a vertical blanking task.

ENABLING THE COLOR DISPLAY

To enable color rather than black and white display, you need to set bit 9 in BPLCON0. Doing so enables the color burst signal on composite video; it does not affect RGB video.

SUMMARY

The steps for defining a basic playfield are summarized below:

1. **Define Playfield Characteristics**
   - a. Specify height in lines:
     - o 200 maximum for non-interlaced mode.
     - o 400 maximum for interlaced mode.
   - b. Specify width in pixels:
     - o 320 maximum for low-resolution mode.
     - o 640 maximum for high-resolution mode.
   - c. Specify color for each pixel:
     - o Load desired colors in color table registers.
     - o Define color of each pixel in terms of the binary value that points at the desired color register.
     - o Build bit-planes.
     - o Set bit-plane registers:
       - * Bits 12-14 in BPLCON0 - number of bit-planes (BPU2 - BPU0).
       - * BPLxPTH - pointer to bit-plane starting position in memory (written as a long word).
   - d. Specify resolution:
     - o Low resolution:
       - * 320 pixels in each horizontal line.
       - * Clear bit 15 in register BPLCON0 (HIRES).

        - o High resolution:
            - * 640 pixels in each horizontal line.
            - * Set bit 15 in register BPLCON0 (HIRES).

    e. Specify interlaced or non-interlaced mode:

        - o Interlaced mode:
            - * 400 vertical lines.
            - * Set bit 2 in register BPLCON0 (LACE).
        - o Non-interlaced mode:
            - * 200 vertical lines.
            - * Clear bit 2 in BPLCON0 (LACE).

2. **Allocate Memory.** To calculate data-bytes in the total bit-planes, use the following formula:

    Bytes per line * lines in playfield * number of bit-planes

3. **Define Size of Display Window.**
    - o Write start position of display window in DIWSTRT:
        - * Horizontal position in bits 0 through 7 (low-order bits).
        - * Vertical position in bits 8 through 15 (high-order bits).
    - o Write stop position of display window in DIWSTOP:
        - * Horizontal position in bits 0 through 7.
        - * Vertical position in bits 8 through 15.

4. **Define Data Fetch.** Set registers DDFSTRT and DDFSTOP:
    - o For DDFSTRT, use the horizontal position as shown in "Setting the Display Window Starting Position."
    - o For DDFSTOP, use the horizontal position as shown in "Setting the Display Window Stopping Position."

5. **Define Modulo.** Set registers BPL1MOD and BPL2MOD. Set modulo to 0 for non-interlaced, 40 for interlaced.

6. **Write Copper Instructions To Handle Redisplay.**

7. **Enable Color Display.** Set bit 9 in BPLCON0 to enable the color display on a composite video monitor. RGB video is not affected.

EXAMPLES OF FORMING BASIC PLAYFIELDS

The following examples show how to set the registers and write the coprocessor lists for two different playfields.

The first example sets up a 320 x 200 playfield with one bit-plane, which is located at $21000. Also, a Copper list is set up at $20000.

```
CUSTOM    EQU  $DFF000
BPLCON0   EQU  $100
BPLCON1   EQU  $102
BPLCON2   EQU  $104
BPL1MOD   EQU  $108
DDFSTRT   EQU  $092
DDFSTOP   EQU  $094
DIWSTRT   EQU  $08E
DIWSTOP   EQU  $090
VPOSR     EQU  $004
COLOR00   EQU  $180
COLOR01   EQU  $182
COLOR02   EQU  $184
COLOR03   EQU  $186
DMACON    EQU  $096
COP1LCH   EQU  $080                ;Copper location register 1
                                   ;  (high three bits)
LEA       CUSTOM,A0                ;A0 points at custom chips
MOVE.W    #$1200,BPLCON0(A0)       ;One bit-plane, enable composite color
MOVE.W    #0,BPLCON1(A0)           ;Set horizontal scroll value to 0
MOVE.W    #0,BPL1MOD(A0)           ;Set modulo to 0 for all odd bit-planes
MOVE.W    #$0038,DDFSTRT(A0)       ;Set data-fetch start to $38
MOVE.W    #$00D0,DDFSTOP(A0)       ;Set data-fetch stop to $D0
MOVE.W    #$2C81,DIWSTRT(A0)       ;Set DIWSTRT to $2C81
MOVE.W    #$F4C1,DIWSTOP(A0)       ;Set DIWSTOP to $F4C1
MOVE.W    #$0F00,COLOR00(A0)       ;Set background color to red
MOVE.W    #$0FF0,COLOR01(A0)       ;Set color register 1 to yellow
```

```
;
; Fill bit-plane with $FF00FF00 to produce stripes
;
         MOVE.L     #$21000,A1               ;Point at beginning of bit-plane
         MOVE.L     #$FF00FF00,D0            ;We will write $FF00FF00 long words
         MOVE.W     #2000,D1                 ;2000 long words = 8000 bytes
;
LOOP:    MOVE.L     D0,(A1)+                 ;Write a long word
         SUBQ.W     #1,D1                    ;Decrement counter
         BNE        LOOP                     ;Loop until bit-plane is filled
;
; Set up Copper list at $20000
;
         MOVE.L     #$20000,A1                ;Point at Copper list destination
         LEA        COPPERL,A2                ;Point A2 at Copper list data
CLOOP:   MOVE.L     (A2),(A1)+                ;Move a word
         CMPI.L     #$FFFFFFFE,(A2)+          ;Check for last longword of Copper list
         BNE        CLOOP                     ;Loop until entire copper list is moved
;
; Point Copper at Copper list
;
         MOVE.L     #$20000,COP1LCH(A0)       ;Write to Copper location register
         MOVE.W     COPJMP1(A0),D0            ;Force copper to $20000
;
; Start DMA
;
         MOVE.W     #$8380,DMACON(A0)         ;Enable bit-plane and Copper DMA
         BRA        ....                      ;Go do next task
;
; This is the data for the Copper list.
;
COPPERL:
         DC.W       $00E0,$0002               ;Move $0002 to address $0E0
                                              ;  (BPL1PTH)
         DC.W       $00E2,$1000               ;Move $1000 to address $0E2
                                              ;  (BPL1PTL)
         DC.W       $FFFF,$FFFE               ;End of Copper list
;
```

The second example sets up a high-resolution, interlaced display with one bit-plane. The equates are the same as the previous example so they aren't repeated here.

```
         LEA      CUSTOM,A0                 ;Address of custom chips
```

```
        MOVE.W  #$9204,BPLCON0(A0)      ;Hires, one bit-plane, interlaced
        MOVE.W  #0,BPLCON1(A0)          ;Horizontal scroll value = 0
        MOVE.W  #80,BPL1MOD(A0)         ;Modulo = 80 for odd bit-planes
        MOVE.W  #80,BPL2MOD(A0)         ;Ditto for even bit-planes
        MOVE.W  #$003C,DDFSTRT(A0)      ;Set data-fetch start for hires
        MOVE.W  #$00D4,DDFSTOP(A0)      ;Set data-fetch stop
        MOVE.W  #$2C81,DIWSTRT(A0)      ;Set display window start
        MOVE.W  #$F4C1,DIWSTOP(A0)      ;Set display window stop
;
; Set up color registers
;
        MOVE.W  #$000F,COLOR00(A0)      ;Background color = blue
        MOVE.W  #$0FFF,COLOR01(A0)      ;Foreground color = white
;
; Set up bit-plane at $20000
;
        LEA     $20000,A1               ;Point A1 at bit-plane
        LEA     CHARLIST,A2             ;A2 points at character data
        MOVE.W  #400,D1                 ;Write 400 lines of data
        MOVE.W  #20,D0                  ;Write 20 long words per line
L1:
        MOVE.L  (A2),(A1)+              ;Write a long word
        SUBQ.W  #1,D0                   ;Decrement counter
        BNE     L1                      ;Loop until line is full
;
        MOVE.W  #20,D0                  ;Reset long word counter
        ADDQ.L  #4,A2                   ;Point at next word in char list
        CMPI.L  #$FFFFFFFF,(A2)         ;End of char list?
        BNE     L2
        LEA     CHARLIST,A2             ;Yes, reset A2 to beginning of list
L2:
        SUBQ.W  #1,D1                   ;Decrement line counter
        BNE     L1                      ;Loop until all lines are full
;
; Start DMA
;
        MOVE.W  #$8300,DMACON(A0)       ;Enable bit-plane DMA only,
                                        ;  no Copper
```

Because this example has no Copper list, it sits in a loop waiting for the vertical blanking interval. When it comes, you check the LOF ("long frame") bit in VPOSR. If LOF = 0, this is a short frame and the bit-plane pointers are set to point to $20050. If LOF = 1, then this is a long frame and the bit-plane pointers are set to point to $20000. This keeps the long and short frames in the right relationship to each other.

```
VLOOP:  MOVE.W   INTREQR(A0),D0          ;Read interrupt requests
        AND.W    #$0020,D0               ;Mask off all but vertical blank
        BEQ      VLOOP                   ;Loop until vertical blank comes
        MOVE.W   #$0020,INTREQ(A0)       ;Reset vertical interrupt
        MOVE.W   VPOSR(A0),D0            ;Read LOF bit into D0 bit 15
        BPL      VL1                     ;If LOF = 0, jump
        MOVE.L   #$20000,BPL1PTH(A0)     ;LOF = 1, point to $20000
        BRA      VLOOP                   ;Back to top
VL1:
        MOVE.L   #$20050,BPL1PTH(A0)     ;LOF = 0, point to $20050
        BRA      VLOOP                   ;Back to top
;
; Character list
;
        DC.L     $18FC3DF0,$3C6666D8,$3C66C0CC,$667CC0CC
        DC.L     $7E66C0CC,$C36666D8,$C3FC3DF0,$00000000
        DC.L     $FFFFFFFF
```

Forming a Dual-playfield Display

For more flexibility in designing your background display, you can specify two playfields instead of one. In dual-playfield mode, one playfield is displayed directly in front of the other. For example, a computer game display might have some action going on in one playfield in the background, while the other playfield is showing a control panel in the foreground. You can then change either the foreground or the background without having to redesign the entire display. You can also move the two playfields independently.

A dual-playfield display is similar to a single-playfield display, differing only in these aspects:

- Each playfield in a dual display is formed from one, two or three bit-planes.
- The colors in each playfield (up to seven plus transparent) are taken from different sets of color registers.
- You must set a bit to activate dual-playfield mode.

Figure 3-12 shows a dual-playfield display.

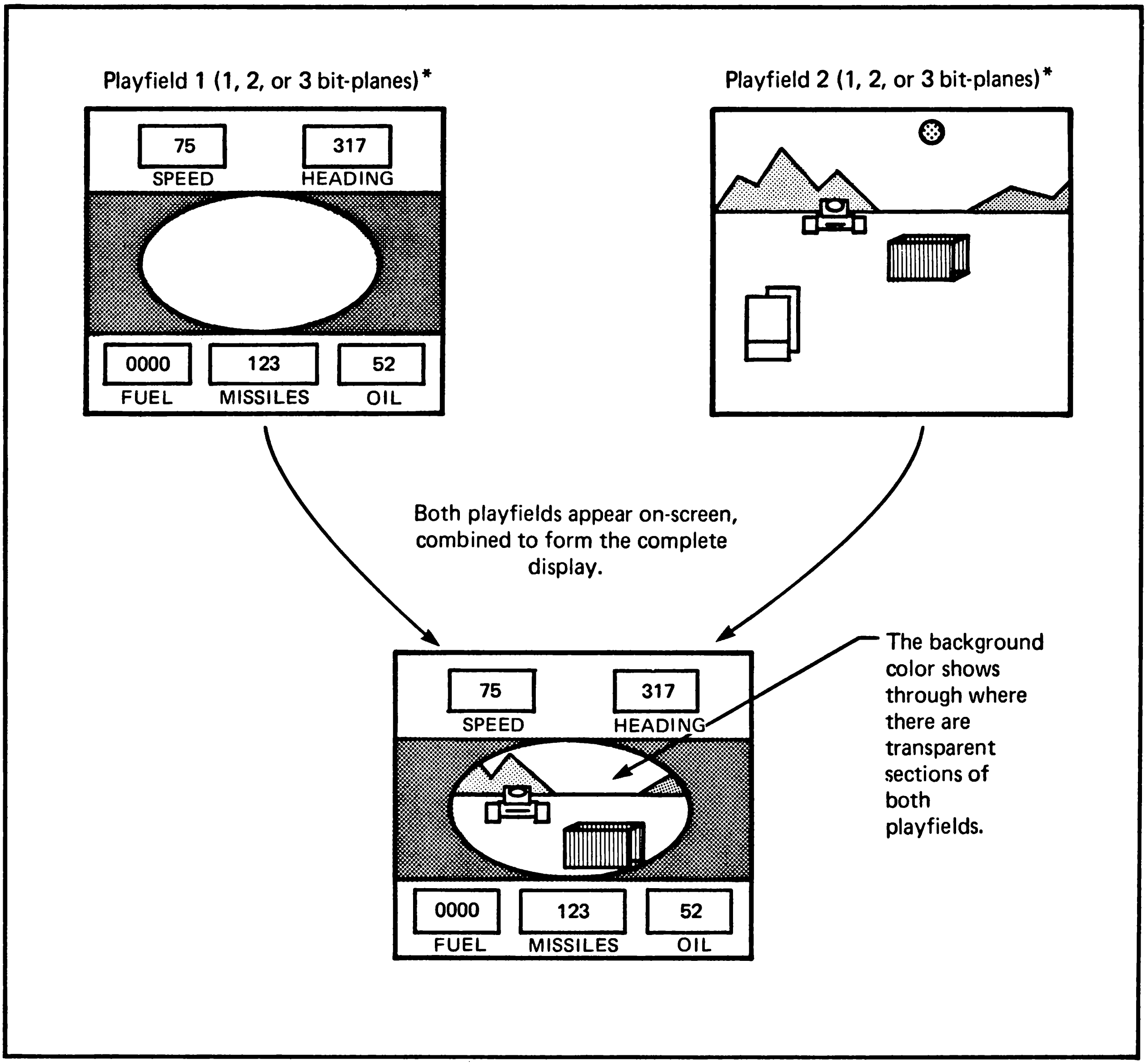

Playfield 1 (1, 2, or 3 bit-planes)*
75
SPEED
317
HEADING
0000
FUEL
123
MISSILES
52
OIL
Playfield 2 (1, 2, or 3 bit-planes)*
Both playfields appear on-screen, combined to form the complete display.
75
SPEED
317
HEADING
0000
FUEL
123
MISSILES
52
OIL
The background color shows through where there are transparent sections of both playfields.

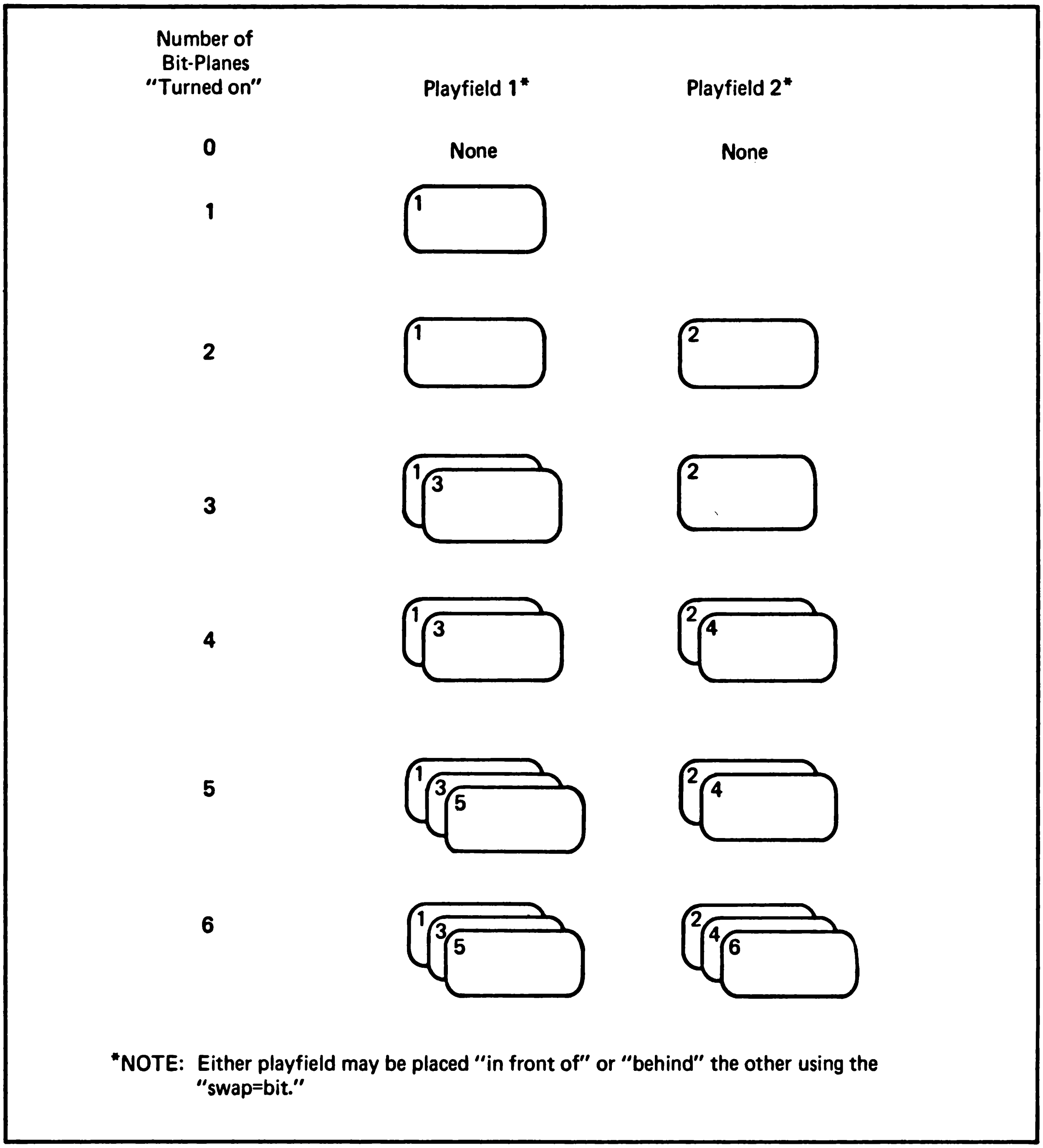
Figure 3-12: A Dual-playfield Display

In figure 3-12, one of the colors in each playfield is "transparent" (color 0 in playfield 1 and color 8 in playfield 2). You can use transparency to allow selected features of the background playfield to show through.

In dual-playfield mode, each playfield is formed from up to three bit-planes. Color registers 0 through 7 are assigned to playfield 1, depending upon how many bit-planes you use. Color registers 8 through 15 are assigned to playfield 2.

Bit-Plane Assignment in Dual-playfield Mode

The three odd-numbered bit-planes (1, 3, and 5) are grouped together by the hardware and may be used in playfield 1. Likewise, the three even-numbered bit-planes (2, 4, and 6) are grouped together and may be used in playfield 2. The bit-planes are assigned alternately to each playfield, as shown in figure 3-13. Note that in high-resolution mode, you can have up to two bit-planes in each playfield — bit-planes 1 and 3 in playfield 1 and bit-planes 2 and 4 in playfield 2.


Number of Bit-Planes "Turned on"
Playfield 1*
Playfield 2*
0
None
None
1
1
2
1
2
3
1
3
2
4
1
3
2
4
5
1
3
5
2
4
6
1
3
5
2
4
6

*NOTE: Either playfield may be placed "in front of" or "behind" the other using the "swap=bit."

Figure 3-13: How Bit-Planes Are Assigned to Dual Playfields

COLOR REGISTERS IN DUAL-PLAYFIELD MODE

When you are using dual playfields, the hardware interprets color numbers for playfield 1 from the bit combinations of bit-planes 1, 3, and 5. Bits from PLANE 5 have the highest significance and form the most significant digit of the color register number. Bits from PLANE 0 have the lowest significance. These bit combinations select the first eight color registers from the color palette as shown in table 3-7.

Table 3-7: Playfield 1 Color Registers — Low-resolution Mode

PLAYFIELD 1

Bit Combination	**Color Selected**
000	Transparent mode
001	COLOR1
010	COLOR2
011	COLOR3
100	COLOR4
101	COLOR5
110	COLOR6
111	COLOR7

The hardware interprets color numbers for playfield 2 from the bit combinations of bit-planes 2, 4, and 6. Bits from PLANE 6 have the highest significance. Bits from PLANE 2 have the lowest significance. These bit combinations select the color registers from the second eight colors in the color table as shown in table 3-8.

Table 3-8: Playfield 2 Color Registers — Low-resolution Mode

PLAYFIELD 2

Bit Combination	**Color Selected**
000	Transparent mode
001	COLOR9
010	COLOR10
011	COLOR11
100	COLOR12
101	COLOR13
110	COLOR14
111	COLOR15

Combination 000 selects transparent mode, to show the color of whatever object (the other playfield, a sprite, or the background color) may be "behind" the playfield.

Table 3-9 shows the color registers for high-resolution, dual-playfield mode.

Table 3-9: Playfields 1 and 2 Color Registers — High-resolution Mode

PLAYFIELD 1

Bit Combination	**Color Selected**
00	Transparent mode
01	COLOR1
10	COLOR2
11	COLOR3

PLAYFIELD 2

Bit Combination	**Color Selected**
00	Transparent mode
01	COLOR9
10	COLOR10
11	COLOR11

DUAL-PLAYFIELD PRIORITY AND CONTROL

Either playfield 1 or 2 may have priority; that is, either one may be displayed in front of the other, although playfield 1 normally has priority. The bit known as PF2PRI (bit 6) in register BPLCON2 is used to control priority. When PF2PRI = 1, playfield 2 has priority over playfield 1. When PF2PRI = 0, playfield 1 has priority.

You can also control the relative priority of playfields and sprites. Chapter 7, "System Control Hardware," shows you how to control the priority of these objects.

You can control the two playfields separately as follows:

- They can have different-sized representations in memory, and different portions of each one can be selected for display.
- They can be scrolled separately.

NOTE:

> You must take special care when scrolling one playfield and holding the other stationary. When you are scrolling low-resolution playfields, you must fetch one word more than the width of the playfield you are trying to scroll (two words more in high-resolution mode) in order to provide some data to display when the actual scrolling takes place. Only one data-fetch start register and one data-fetch stop register are available, and these are shared by both playfields. If you want to scroll one playfield and hold the other, you must adjust the data-fetch start and data-fetch stop to handle the playfield being scrolled. Then, you must adjust the modulo and the bit-plane pointers of the playfield that is *not* being scrolled to maintain its position on the display. In low-resolution mode, you adjust the pointers by -2 and the modulo by -2. In high-resolution mode, you adjust the pointers by -4 and the modulo by -4.

ACTIVATING DUAL-PLAYFIELD MODE

Writing a 1 to bit 10 (called DBLPF) of the bit-plane control register BPLCON0 selects dual-playfield mode. Selecting dual-playfield mode changes both the way the hardware groups the bit-planes for color interpretation—all odd-numbered bit-planes are grouped together and all even-numbered bit-planes are grouped together—and the way hardware can move the bit-planes on the screen.

SUMMARY

The steps for defining dual playfields are almost the same as those for defining the basic playfield. Only in the following steps does the dual-playfield creation process differ from that used for the basic playfield:

- **Loading colors into the registers.** Keep in mind that color registers 0-7 are used by playfield 1 and registers 8 through 15 are used by playfield 2 (if there are three bit-planes in each playfield).
- **Building bit-planes.** Recall that playfield 1 is formed from PLANES 1, 3, and 5 and playfield 2 from PLANES 2, 4, and 6.
- **Setting the modulo registers.** Write the modulo to both BPL1MOD and BPL2MOD as you will be using both odd- and even-numbered bit-planes.

These steps are added:

- **Defining priority.** If you want playfield 2 to have priority, set bit 6 (PF2PRI) in BPLCON2 to 1.
- **Activating dual-playfield mode.** Set bit 10 (DBLPF) in BPLCON0 to 1.

Bit-planes and Display Windows of All Sizes

You have seen how to form single and dual playfields in which the playfield in memory is the same size as the display window. This section shows you how to define and use a playfield whose big picture in memory is larger than the display window, how to define display windows that are larger or smaller than the normal playfield size, and how to move the display window in the big picture.

WHEN THE BIG PICTURE IS LARGER THAN THE DISPLAY WINDOW

If you design a memory picture larger than the display window, you must choose which part of it to display. Displaying a portion of a larger playfield differs in the following ways from displaying the basic playfields described up to now:

- If the big picture in memory is larger than the display window, you must respecify the modulos. The modulo must be some value other than 0.
- You must allocate more memory for the larger memory picture.

Specifying the Modulo

For a memory picture wider than the display window, you need to respecify the modulo so that the correct data words are fetched for each line of the display. As an example, assume the display window is the standard 320 pixels wide, so 40 bytes are to be displayed on each line. The big picture in memory, however, is exactly twice as wide as the display window, or 80 bytes wide. Also, assume that you wish to display the left half of the big picture. Figure 3-14 shows the relationship between the big picture and the picture to be displayed.

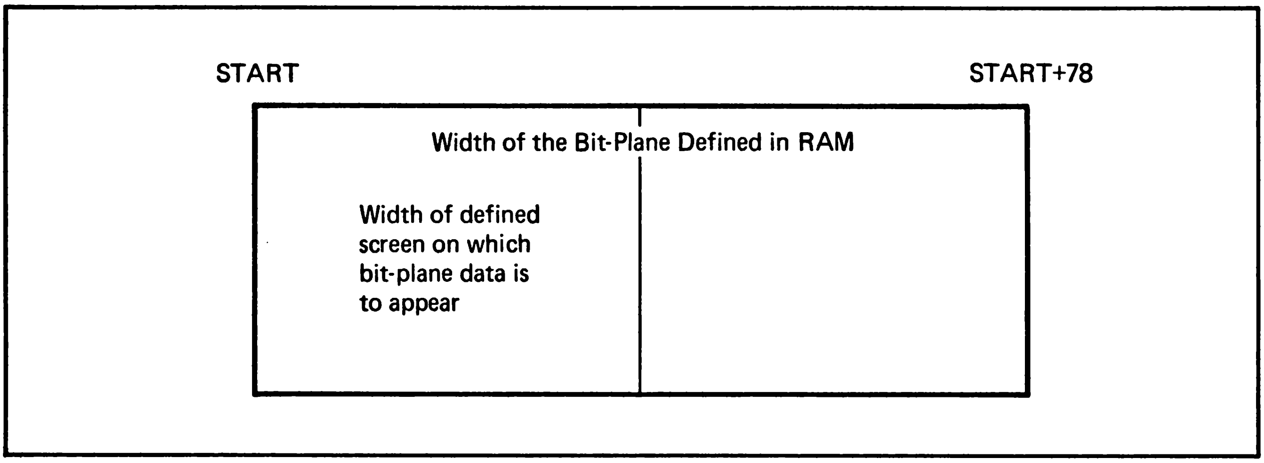

START
START+78
Width of the Bit-Plane Defined in RAM
Width of defined screen on which bit-plane data is to appear

Figure 3-14: Memory Picture Larger than the Display

Because 40 bytes are to be fetched for each line, the data fetch for line 1 is as shown in figure 3-15.

Data for Line 1:					
Location:	START	START+2	START+4	...	START+38
	Leftmost Display Word	Next Word	Next Word		Last Display Word

Screen data fetch stops (DDFSTOP) for each horizontal line after the last word on the line has been fetched.

Figure 3-15: Data Fetch for the First Line When Modulo = 40

At this point, BPLxPTH and BPLxPTL contain the value START+40. The modulo, which is 40, is added to the current value of the pointer so that when it begins the data fetch for the next line, it fetches the data that you intend for that line. The data fetch for line 2 is shown in figure 3-16.

Data for Line 2:					
Location:	START+80	START+82	START+84	...	START+118
	Leftmost Display Word	Next Word	Next Word		Last Display Word

Figure 3-16: Data Fetch for the Second Line When Modulo = 40

To display the right half of the big picture, you set up a vertical blanking routine to start the bit-plane pointers at location START+40 rather than START with the modulo remaining at 40. The data layout is shown in figures 3-17 and 3-18.

Data for Line 2:				
Location:	START+40	START+42	START+44 ...	START+78
	Leftmost Display Word	Next Word	Next Word	Last Display Word

Figure 3-17: Data Layout for First Line—Right Half of Big Picture

Now, the bit-plane pointers contain the value START+80. The modulo (40) is added to the pointers so that when they begin the data fetch for the second line, the correct data is fetched.

Data for Line 2:				
Location:	START+120	START+122	START+124 ...	START+158
	Leftmost Display Word	Next Word	Next Word	Last Display Word

Figure 3-18: Data Layout for Second Line—Right Half of Big Picture

Remember, in high-resolution mode, you need to fetch twice as many bytes as in low-resolution mode. For a normal-sized display, you fetch 80 bytes for each horizontal line instead of 40.

Specifying the Data Fetch

The data-fetch registers specify the beginning and end positions for data placement on each horizontal line of the display. You specify data fetch in the same way as shown in the section called "Forming a Basic Playfield."

Memory Allocation

For larger memory pictures, you need to allocate more memory. Here is a formula for calculating memory requirements in general:

bytes per line * lines in playfield * # of bit-planes

Thus, if the wide playfield described in this section is formed from two bit-planes, it requires:

80 * 200 * 2 = 32,000 bytes of memory

Recall that this is the memory requirement for the playfield alone. You need more memory for any sprites, animation, audio, or application programs you are using.

Selecting the Display Window Starting Position

The display window starting position is the horizontal and vertical coordinates of the upper left-hand corner of the display window. One register, DIWSTRT, holds both the horizontal and vertical coordinates, known as HSTART and VSTART. The eight bits allocated to HSTART are assigned to the first 256 positions, counting from the leftmost possible position. Thus, you can start the display window at any pixel position within this range.

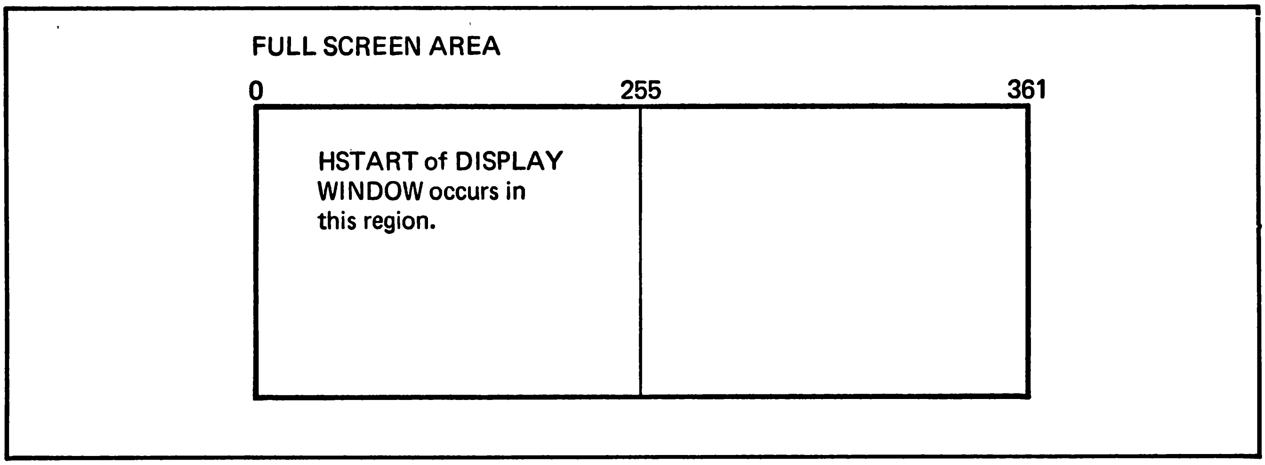

FULL SCREEN AREA
0
255
361
HSTART of DISPLAY WINDOW occurs in this region.

Figure 3-19: Display Window Horizontal Starting Position

The eight bits allocated to VSTART are assigned to the first 256 positions counting down from the top of the display.

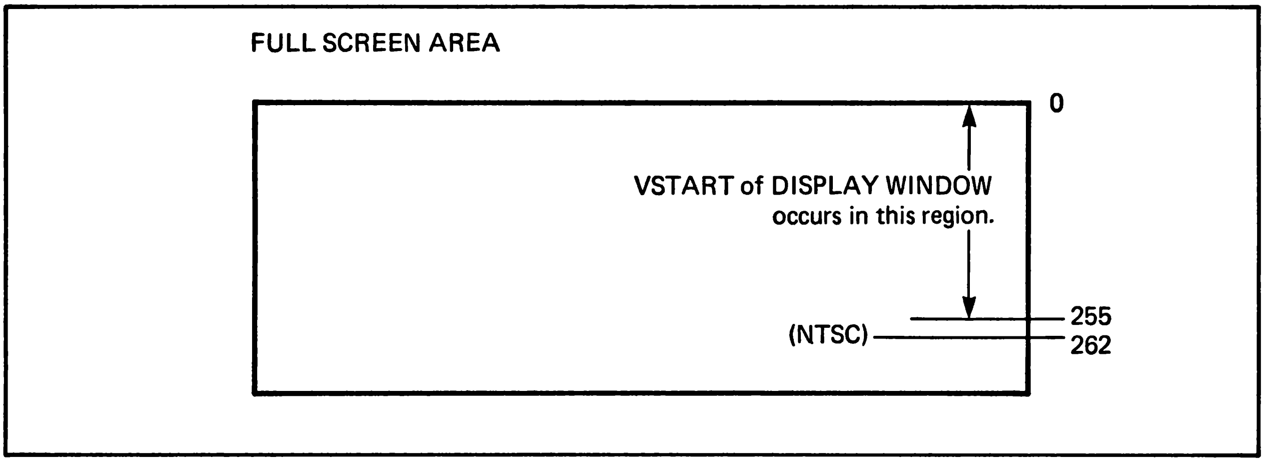

FULL SCREEN AREA
0
VSTART of DISPLAY WINDOW occurs in this region.
(NTSC)
255
262

Figure 3-20: Display Window Vertical Starting Position

Recall that you select the values for the starting position as if the display were in low-resolution, non-interlaced mode. Keep in mind, though, that for interlaced mode the display window should be an even number of lines in height to allow for equal-sized odd and even fields.

To set the display window starting position, write the value for HSTART into bits 0 through 7 and the value for VSTART into bits 8 through 15 of DIWSTRT.

Selecting the Stopping Position

The stopping position for the display window is the horizontal and vertical coordinates of the lower right-hand corner of the display window. One register, DIWSTOP, contains both coordinates, known as HSTOP and VSTOP.

See the notes in the "Forming a Basic Playfield" section for instructions on setting these registers.

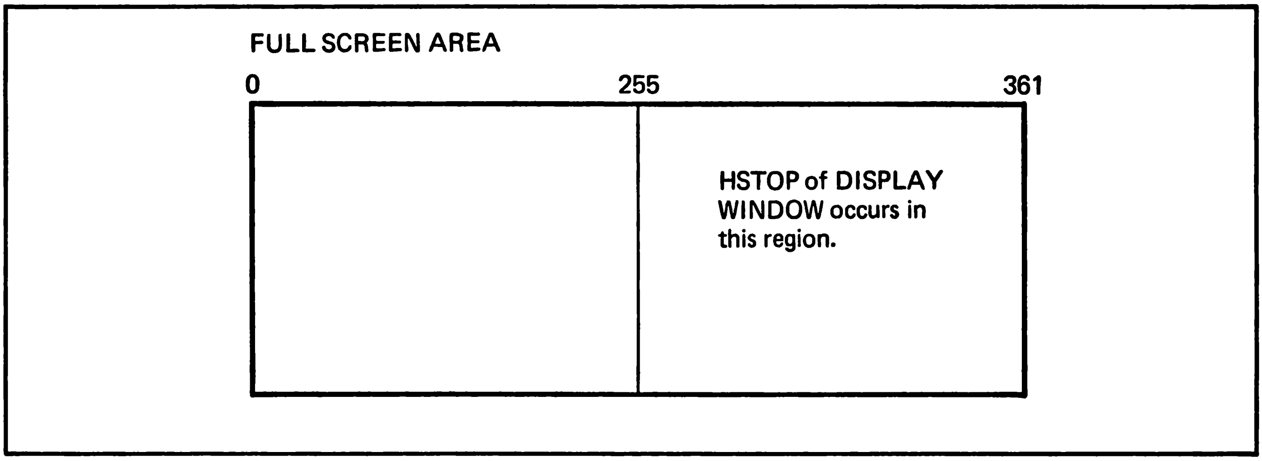


Figure 3-21: Display Window Horizontal Stopping Position

Select a value that represents the correct position in low-resolution, non-interlaced mode.

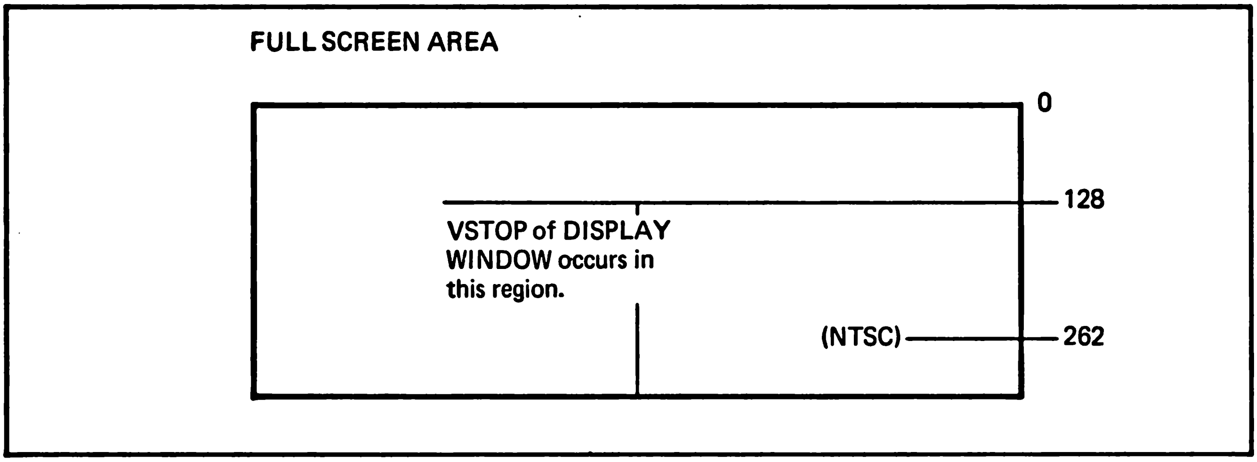


Figure 3-22: Display Window Vertical Stopping Position

To set the display window stopping position, write HSTOP into bits 0 through 7 and VSTOP into bits 8 through 15 of DIWSTOP.

MAXIMUM DISPLAY WINDOW SIZE

The maximum size of a playfield display is determined by the maximum number of lines and the maximum number of columns. Vertically, the restrictions are simple. No data can be displayed in the vertical blanking area, which ranges from line 0 through line 19 (20 lines total). This leaves 242 lines of displayable screen video (interlaced mode doubles this to 484).

Horizontally, the situation is similar. Strictly speaking, the hardware sets a rightmost limit to DDFSTOP of ($D8) and a leftmost limit to DDFSTRT of ($18). This gives a maximum of 25 words fetched in low-resolution mode. In high-resolution mode the maximum here is 49 words, because the rightmost limit remains ($D8) and only one word is fetched at this limit. However, horizontal blanking actually limits the displayable video to 376 low-resolution pixels (23.5 words). In addition, it should be noted that using a data-fetch start earlier than ($38) will disable some sprites.

Moving (Scrolling) Playfields

If you want a background display that moves, you can design a playfield larger than the display window and scroll it. If you are using dual playfields, you can scroll them separately.

In vertical scrolling, the playfield appears to move smoothly up or down on the screen. All you need do for vertical scrolling is progressively increase or decrease the starting address for the bit-plane pointers by the size of a horizontal line in the playfield. This has the effect of showing a lower or higher part of the picture each field time.

In horizontal scrolling the playfield appears to move from right to left or left to right on the screen. Horizontal scrolling works differently from vertical scrolling — you must arrange to fetch one more word of data for each display line and delay the display of this data.

For either type of scrolling, resetting of pointers or data-fetch registers can be handled by the Copper during the vertical blanking interval.

VERTICAL SCROLLING

You can scroll a playfield upward or downward in the window. Each time you display the playfield, the bit-plane pointers start at a progressively higher or lower place in the big picture in memory. As the value of the pointer increases, more of the lower part of the picture is shown and the picture appears to scroll upward. As the value of the pointer decreases, more of the upper part is shown and the picture scrolls downward. If your picture has 200 vertical lines, each step can be as little as 1/200th of the screen. In interlaced mode each step could be 1/400th of the screen if clever manipulation of the pointers is used, but it is recommended that scrolling be done two lines at a time to maintain the odd/even field relationship.

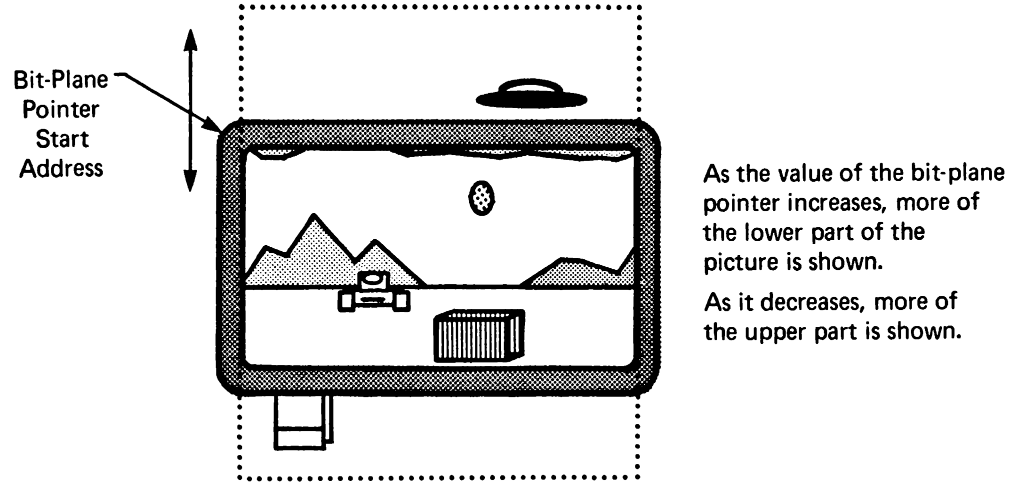

Bit-Plane Pointer Start Address
As the value of the bit-plane pointer increases, more of the lower part of the picture is shown.
As it decreases, more of the upper part is shown.

Figure 3-23: Vertical Scrolling

To set up a playfield for vertical scrolling, you need to form bit-planes tall enough to allow for the amount of scrolling you want, write software to calculate the bit-plane pointers for the scrolling you want, and allow for the Copper to use the resultant pointers.

Assume you wish to scroll a playfield upward one line at a time. To accomplish this, before each field is displayed, the bit-plane pointers have to increase by enough to ensure that the pointers begin one line lower each time. For a normal-sized, low-resolution display in which the modulo is 0, the pointers would be incremented by 40 bytes each time.

HORIZONTAL SCROLLING

You can scroll playfields horizontally from left to right or right to left on the screen. You control the speed of scrolling by specifying the amount of delay in pixels. Delay means that an extra word of data is fetched but not immediately displayed. The extra word is placed just to the left of the window's leftmost edge and before normal data fetch. As the display shifts to the right, the bits in this extra word appear on-screen at the left-hand side of the window as bits on the right-hand side disappear off-screen. For each pixel of delay, the on-screen data shifts one pixel to the right each display field. The greater the delay, the greater the speed of scrolling. You can have up to 15 pixels of delay. In high-resolution mode, scrolling is in increments of 2 pixels. Figure 3-24 shows how the delay and extra data fetch combine to cause the scrolling effect.

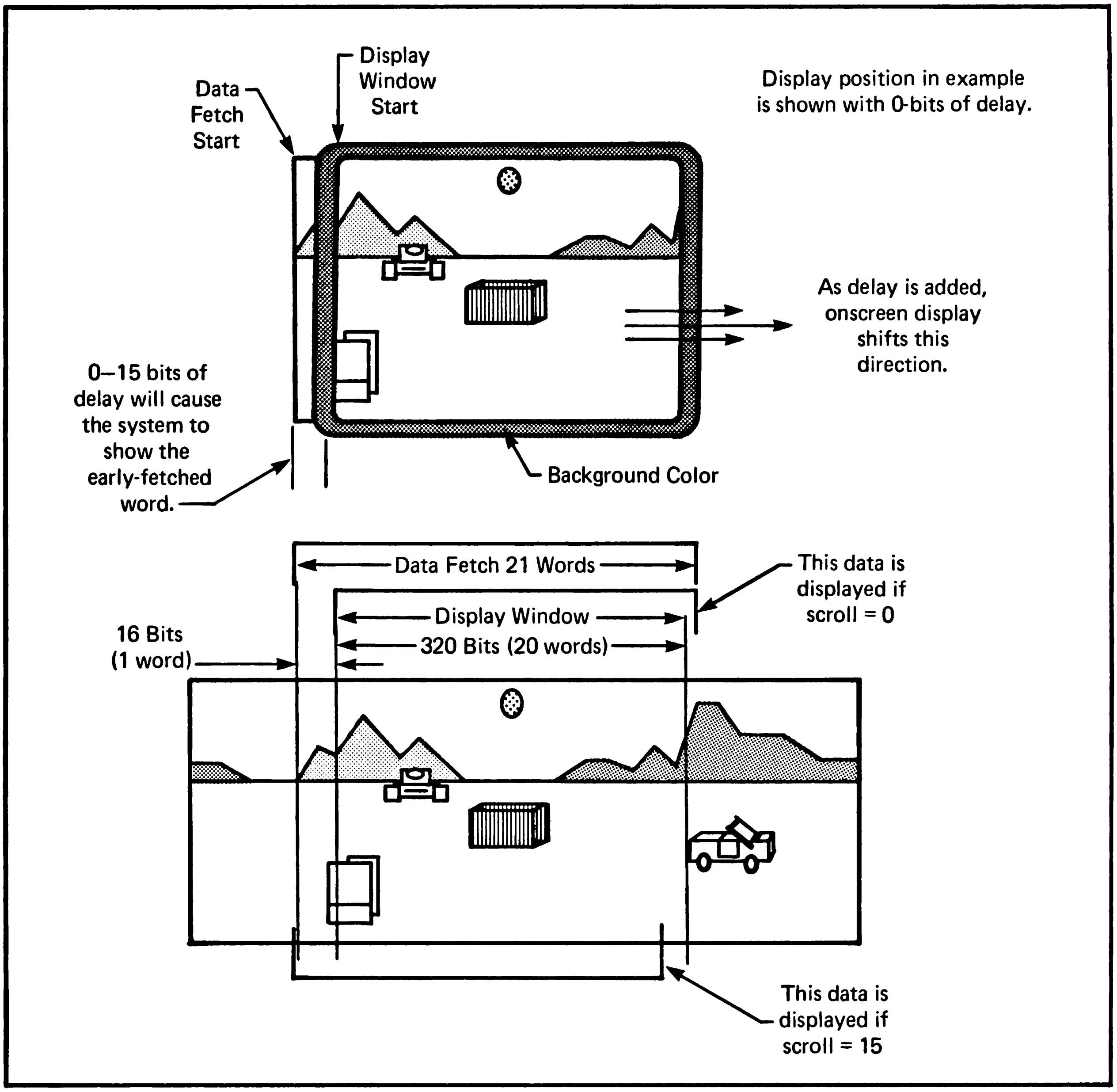

Display Window Start
Data Fetch Start
Display position in example is shown with 0-bits of delay.
As delay is added, onscreen display shifts this direction.
0–15 bits of delay will cause the system to show the early-fetched word.
Background Color
Data Fetch 21 Words
This data is displayed if scroll = 0
Display Window
320 Bits (20 words)
16 Bits (1 word)
This data is displayed if scroll = 15

Figure 3-24: Horizontal Scrolling

Note that fetching an extra word for scrolling will disable some sprites.

To set up a playfield for horizontal scrolling, you need to

- Define bit-planes wide enough to allow for the scrolling you need.
- Set the data-fetch registers to correctly place each horizontal line, including the extra word, on the screen.
- Set the delay bits.
- Set the modulo so that the bit-plane pointers begin at the correct word for each line.
- Write Copper instructions to handle the changes during the vertical blanking interval.

Specifying Data Fetch in Horizontal Scrolling

The normal data-fetch start for non-scrolled displays is ($38). If horizontal scrolling is desired, then the data fetch must start one word sooner (DDFSTRT = $0030). Incidentally, this will disable sprite 7. DDFSTOP remains unchanged. Remember that the settings of the data-fetch registers affect both playfields.

Specifying the Modulo in Horizontal Scrolling

As always, the modulo is two counts less than the difference between the address of the next word you want to fetch and the address of the last word that was fetched. As an example for horizontal scrolling, let us assume a 40-byte display in an 80-byte "big picture." Because horizontal scrolling requires a data fetch of two extra bytes, the data for each line will be 42 bytes long.

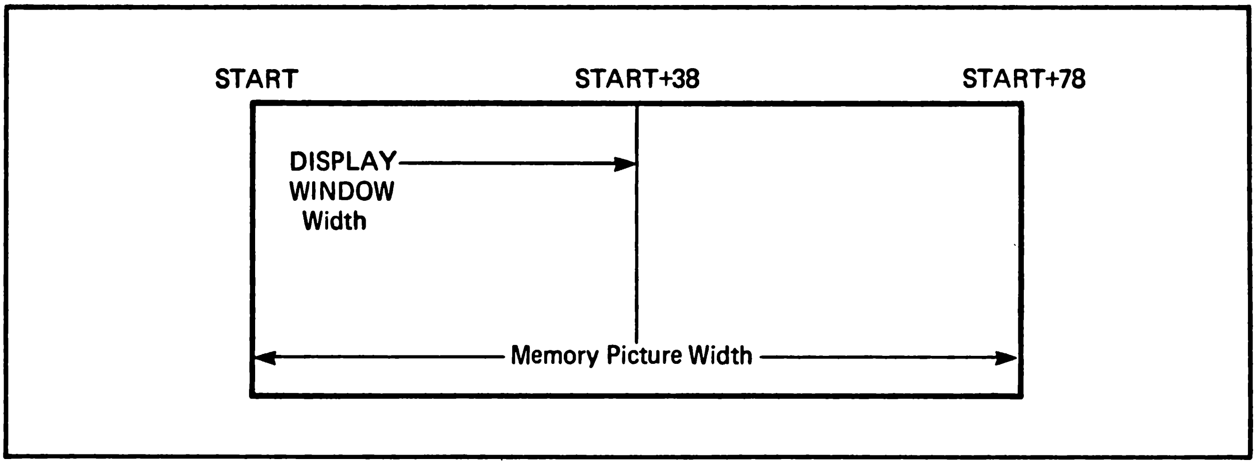


Figure 3-25: Memory Picture Larger Than the Display Window

Data for Line 1:

Location:	START	START+2	START+4	... START+40
	Leftmost Display Word	Next Word	Next Word	Last Display Word

Figure 3-26: Data for Line 1 - Horizontal Scrolling

At this point, the bit-plane pointers contain the value START+42. Adding the modulo of 38 gives the correct starting point for the next line.

Data for Line 2:

Location:	START+80	START+82	START+84	... START+120
	Leftmost Display Word	Next Word	Next Word	Last Display Word

Figure 3-27: Data for Line 2—Horizontal Scrolling

In the BPLxMOD registers you set the modulo for each bit-plane used.

Specifying Amount of Delay

The amount of delay in horizontal scrolling is controlled by bits 7-0 in BPLCON1. You set the delay separately for each playfield; bits 3-0 for playfield 1 (bit-planes 1, 3, and 5) and bits 7-4 for playfield 2 (bit-planes 2, 4, and 6).

NOTE: Always set all six bits, even if you have only one playfield. Set 3-0 and 7-4 to the same value if you are using only one playfield.

The following example sets the horizontal scroll delay to 7 for both playfields.

```
BPLCON1    EQU     $DFF102          ;Horizontal scroll register
;
           MOVE.W  #$77,BPLCON1
```

SUMMARY

The steps for defining a scrolled playfield are the same as those for defining the basic playfield, except for the following steps:

- **Defining the data fetch.** Fetch one extra word per horizontal line and start it 16 pixels before the normal (unscrolled) data-fetch start.
- **Defining the modulo.** The modulo is two counts greater than when there is no scrolling.

These steps are added:

- **For vertical scrolling, reset the bit-plane pointers for the amount of the scrolling increment.** Reset BPLxPTH and BPLxPTL during the vertical blanking interval.
- **For horizontal scrolling, specify the delay.** Set bits 7-0 in BPLCON1 for 0 to 15 bits of delay.

Advanced Topics

This section describes features that are used less often or are optional.

INTERACTIONS AMONG PLAYFIELDS AND OTHER OBJECTS

Playfields share the display with sprites. Chapter 7, "System Control Hardware," shows how playfields can be given different video display priorities relative to the sprites and how playfields can collide with (overlap) the sprites or each other.

HOLD-AND-MODIFY MODE

This is a special mode that allows you to produce up to 4,096 colors on the screen at the same time. Normally, as each value formed by the combination of bit-planes is selected, the data contained in the selected color register is loaded into the color output circuit for the pixel being written on the screen. Therefore, each pixel is colored by the contents of the selected color register.

In hold-and-modify mode, however, the value in the color output circuitry is held, and one of the three components of the color (red, green, or blue) is modified by bits coming from certain preselected bit-planes. After modification, the pixel is written to the screen.

The hold-and-modify mode allows very fine gradients of color or shading to be produced on the screen. For example, you might draw a set of 16 vases, each a different color, using all 16 colors in the color palette. Then, for each vase, you use hold-and-modify to very finely shade or highlight or add a completely different color to each of the vases. Note that a particular hold-and-modify pixel can only change one of the three color values at a time. Thus, the effect has a limited control.

In hold and modify mode, you use all six bit-planes. Planes 5 and 6 are used to modify the way bits from planes 1 - 4 are treated, as follows:

- If the 6-5 bit combination from planes 6 and 5 for any given pixel is 00, normal color selection procedure is followed. Thus, the bit combinations from planes 4 - 1, in that order of significance, are used to choose one of 16 color registers (registers 0 - 15).

  If only five bit-planes are used, the data from the sixth plane is automatically supplied with the value as 0.

- If the 6-5 bit combination is 01, the color of the pixel immediately to the left of this pixel is duplicated and then modified. The bit-combinations from planes 4 - 1 are used to replace the four "blue" bits in the corresponding color register.

- If the 6-5 bit combination is 10, the color of the pixel immediately to the left of this pixel is duplicated and then modified. The bit-combinations from planes 4 - 1 are used to replace the four "red" bits.

- If the 6-5 bit combination is 11, the color of the pixel immediately to the left of this pixel is duplicated and then modified. The bit-combinations from planes 4 - 1 are used to replace the four "green" bits.

Using hold-and-modify mode, it is possible to get by with defining only *one* color register, which is COLOR0, the color of the background. You treat the entire screen as a modification of that original color, according to the scheme above.

Bit 11 of register BPLCON0 selects hold-and-modify mode. The following bits in BPLCON0 must be set for hold-and-modify mode to be active:

- Bit HOMOD, bit 11, is 1.

- Bit DBLPF, bit 10, is 0 (single-playfield mode specified).

- Bit HIRES, bit 15, is 0 (low-resolution mode specified).

- Bits BPU2, BPU1, and BPU0 - bits 14, 13, and 12, are 101 or 110 (five or six bit-planes active).

The following example code generates a six-bit-plane display with hold-and-modify mode turned on. All 32 color registers are loaded with black to prove that the colors are being generated by hold-and-modify. The equates are the usual and are not repeated here.

```
; First, set up the control registers.
;
        LEA      CUSTOM,A0               ;Point A0 at custom chips
        MOVE.W   #$6A00,BPLCON0(A0)      ;Six bit-planes, hold-and-modify mode
        MOVE.W   #0,BPLCON1(A0)          ;Horizontal scroll = 0
        MOVE.W   #0,BPL1MOD(A0)          ;Modulo for odd bit-planes = 0
        MOVE.W   #0,BPL2MOD(A0)          ;Ditto for even bit-planes
        MOVE.W   #$0038,DDFSTRT(A0)      ;Set data-fetch start
        MOVE.W   #$00D0,DDFSTOP(A0)      ;Set data-fetch stop
        MOVE.W   #$2C81,DIWSTRT(A0)      ;Set display window start
        MOVE.W   #$F4C1,DIWSTOP(A0)      ;Set display window stop
;
```

```
; Set all color registers = black to prove that hold-and-modify mode is working.
;
        MOVE.W  #32,D0                  ;Initialize counter
        LEA     CUSTOM+COLOR00,A1       ;Point A1 at first color register
CREGLOOP:
        MOVE.W  #$0000,(A1)+            ;Write black to a color register
        SUBQ.W  #1,D0                   ;Decrement counter
        BNE     CREGLOOP                ;Loop until all color registers set
;
; Fill six bit-planes with an easily recognizable pattern.
;
        MOVE.W  #2000,D0                ;2000 longwords per bit-plane
        MOVE.L  #$21000,A1              ;Point A1 at bit-plane 1
        MOVE.L  #$23000,A2              ;Point A2 at bit-plane 2
        MOVE.L  #$25000,A3              ;Point A3 at bit-plane 3
        MOVE.L  #$27000,A4              ;Point A4 at bit-plane 4
        MOVE.L  #$29000,A5              ;Point A5 at bit-plane 5
        MOVE.L  #$2B000,A6              ;Point A6 at bit-plane 6
FPLLOOP:
        MOVE.L  #$55555555,(A1)+        ;Fill bit-plane 1 with $55555555
        MOVE.L  #$33333333,(A2)+        ;Fill bit-plane 2 with $33333333
        MOVE.L  #$0F0F0F0F,(A3)+        ;Fill bit-plane 3 with $0F0F0F0F
        MOVE.L  #$00FF00FF,(A4)+        ;Fill bit-plane 4 with $00FF00FF
        MOVE.L  #$FFFFFFFF,(A5)+        ;Fill bit-plane 5 with $FFFFFFFF
        MOVE.L  #$00000000,(A6)+        ;Fill bit-plane 6 with $00000000
        SUBQ.W  #1,D0                   ;Decrement counter
        BNE     FPLLOOP                 ;Loop until all bit-planes are full
;
; Set up a Copper list at $20000.
;
        MOVE.L  #$20000,A1              ;Point A1 at Copper list destination
        LEA     COPPERL,A2              ;Point A2 at Copper list image
CLOOP:  MOVE.L  (A2),(A1)+              ;Move a long word
        CMPI.L  #$FFFFFFFE,(A2)+        ;Check for end of Copper list
        BNE     CLOOP                   ;Loop until entire Copper list moved
;
; Point Copper at Copper list.
;
        MOVE.L  #$20000,COP1LCH(A0)     ;Load Copper jump register
        MOVE.W  COPJMP1(A0),D0          ;Force load into Copper P.C.
;
; Start DMA.
;
        MOVE.W  #$8380,DMACON(A0        ;Enable bit-plane and Copper DMA
          BRA       .....next stuff to do.....
;
; Copper list for six bit-planes. Bit-plane 1 is at $21000; 2 is at $23000;
; 3 is at $25000; 4 is at $27000; 5 is at $29000; 6 is at $2B000.
;
COPPERL:
          DC.W      $00E0,$0002               ;Bit-plane 1 pointer = $21000
          DC.W      $00E2,$1000
          DC.W      $00E4,$0002               ;Bit-plane 2 pointer = $23000
          DC.W      $00E6,$3000
          DC.W      $00E8,$0002               ;Bit-plane 3 pointer = $25000
          DC.W      $00EA,$5000
          DC.W      $00EC,$0002               ;Bit-plane 4 pointer = $27000
          DC.W      $00EE,$7000
          DC.W      $00F0,$0002               ;Bit-plane 5 pointer = $29000
          DC.W      $00F2,$9000
          DC.W      $00F4,$0002               ;Bit-plane 6 pointer = $2B000
          DC.W      $00F6,$B000
          DC.W      $FFFF,$FFFE               ;Wait for the impossible, i.e., quit
```

FORMING A DISPLAY WITH SEVERAL DIFFERENT PLAYFIELDS

The graphics library provides the ability to split the screen into several "ViewPorts", each with its own colors and resolutions. See the *Amiga ROM Kernel Manual* for more information.

USING AN EXTERNAL VIDEO SOURCE

An optional board that provides *genlock* is available for the Amiga. Genlock allows you to bring in your graphics display from an external video source (such as a VCR, camera, or laser disk player). When you use genlock, the background color is replaced by the display from this external video source. For more information, see the instructions furnished with the optional board.

SUMMARY OF PLAYFIELD REGISTERS

This section summarizes the registers used in this chapter and the meaning of their bit settings. The color registers are summarized in the next section. See appendix A for a summary of all registers.

BPLCON0 - Bit Plane Control

NOTE: Bits in this register cannot be independently set.

Bit 0 - unused

Bit 1 - ERSY (external synchronization enable)
1 = External synchronization enabled
0 = External synchronization disabled

Bit 2 - LACE (interlace enable)
1 = interlaced mode enabled
0 = non-interlaced mode enabled

Bit 3 - LPEN (light pen enable)

Bits 4-7 not used (make 0)

Bit 8 - GAUD (genlock audio enable)
1 = Genlock audio enabled
0 = Genlock audio disabled

Bit 9 - COLOR_ON (color enable)
1 = composite video color-burst enabled
0 = composite video color-burst disabled

Bit 10 - DBLPF (double-playfield enable)
1 = dual playfields enabled
0 = single playfield enabled

Bit 11 - HOMOD (hold-and-modify enable)
1 = hold-and-modify enabled
0 = hold-and-modify disabled

Bits 14, 13, 12 - BPU2, BPU1, BPU0
Number of bit-planes used.

000 = only a background color
001 = 1 bit-plane, PLANE 1
010 = 2 bit-planes, PLANES 1 and 2
011 = 3 bit-planes, PLANES 1 - 3
100 = 4 bit-planes, PLANES 1 - 4
101 = 5 bit-planes, PLANES 1 - 5
110 = 6 bit-planes, PLANES 1 - 6
111 not used

Bit 15 - HIRES (high-resolution enable)
1 = high-resolution mode
0 = low-resolution mode

BPLCON1 - Bit-plane Control

Bits 3-0 - PF1H(3-0)
Playfield 1 delay

Bits 7-4 - PF2H(3-0)
Playfield 2 delay

Bits 15-8 not used

BPLCON2 - Bit-plane Control

Bit 6 - PF2PRI

1 = Playfield 2 has priority

0 = Playfield 1 has priority

Bits 0-5 Playfield sprite priority

Bits 7-15 not used

DDFSTRT - Data-fetch Start
(Beginning position for data fetch)

Bits 15-8 - not used

Bits 7-3 - pixel position H8-H4

Bits 2-0 - not used

DDFSTOP - Data-fetch Stop
(Ending position for data fetch)

Bits 15-8 - not used

Bits 7-3 - pixel position H8-H4

Bits 1-0 - not used

BPLxPTH - Bit-plane Pointer
(Bit-plane pointer high word, where x is the bit-plane number)

BPLxPTL - Bit-plane Pointer
(Bit-plane pointer low word, where x is the bit-plane number)

DIWSTRT - Display Window Start
(Starting vertical and horizontal coordinates)

Bits 15-8 - VSTART (V7-V0)

Bits 7-0 - HSTART (H7-H0)

DIWSTOP - Display Window Stop
(Ending vertical and horizontal coordinates)

Bits 15-8 - VSTOP (V7-V0)

Bits 7-0 - HSTOP (H7-H0)

BPL1MOD - Bit-plane Modulo
(Odd-numbered bit-planes, playfield 1)

BPL2MOD - Bit-plane Modulo
(Even-numbered bit-planes, playfield 2)

Summary of Color Selection

This section contains summaries of playfield color selection including color register contents, example colors, and the differences in color selection in high-resolution and low-resolution modes.

COLOR REGISTER CONTENTS

Table 3-10 shows the contents of each color register. All color registers are write-only.

Table 3-10: Color Register Contents

Bits	Contents
15 - 12	(Unused)
11 - 8	Red
7 - 4	Green
3 - 0	Blue

SOME SAMPLE COLOR REGISTER CONTENTS

Table 3-11 shows a variety of colors and the hexadecimal values to load into the color registers for these colors.

Table 3-11: Some Register Values and Resulting Colors

Value	**Color**	**Value**	**Color**
$FFF	White	$1FB	Light aqua
$D00	Brick red	$6FE	Sky blue
$F00	Red	$6CE	Light blue
$F80	Red-orange	$00F	Blue
$F90	Orange	$61F	Bright blue
$FB0	Golden orange	$06D	Dark blue
$FD0	Cadmium yellow	$91F	Purple
$FF0	Lemon yellow	$C1F	Violet
$BF0	Lime green	$F1F	Magenta
$8E0	Light green	$FAC	Pink
$0F0	Green	$DB9	Tan
$2C0	Dark green	$C80	Brown
$0B1	Forest green	$A87	Dark brown
$0BB	Blue green	$CCC	Light grey
$0DB	Aqua	$999	Medium grey
		$000	Black

COLOR SELECTION IN LOW-RESOLUTION MODE

Table 3-12 shows playfield color selection in low-resolution mode. If the bit-combinations from the playfields are as shown, the color is taken from the color register number indicated.

Table 3-12: Low-resolution Color Selection

Single Playfield		Dual Playfields	
Normal Mode (Bit-planes 5,4,3,2,1)	**Hold-and-modify Mode (Bit-planes 4,3,2,1)**		**Color Register Number**
		Playfield 1 Bit-planes 5,3,1	
00000	0000	000	0 *
00001	0001	001	1
00010	0010	010	2
00011	0011	011	3
00100	0100	100	4
00101	0101	101	5
00110	0100	110	6
00111	0111	111	7
		Playfield 2 Bit-planes 6,4,2	
01000	1000	000 **	8
01001	1001	001	9
01010	1010	010	10
01011	1011	011	11
01100	1100	100	12
01101	1101	101	13
01110	1110	110	14
01111	1111	111	15
10000	|	|	16
10001	|	|	17
10010	|	|	18
10011	|	|	19
10100	NOT	NOT	20
10101	USED	USED	21
10110	IN	IN	22
10111	THIS	THIS	23
11000	MODE	MODE	24
11001	|	|	25
11010	|	|	26
11011	|	|	27
11100	|	|	28
11101	|	|	29
11110	|	|	30
11111	|	|	31

---

* Color register 0 always defines the background color.

** Selects "transparent" mode instead of selecting color register 8.

COLOR SELECTION IN HOLD-AND-MODIFY MODE

In hold-and-modify mode, the color register contents are changed as shown in table 3-13. This mode is in effect only if bit 10 of BPLCON0 = 1.

Table 3-13: Color Selection in Hold-and-modify Mode

Bit-plane 6	**Bit-plane 5**	**Result**	
0	0	Normal operation	(use color register itself)
0	1	Hold green and red	B = Bit-plane 4-1 contents
1	0	Hold green and blue	R = Bit-plane 4-1 contents
1	1	Hold blue and red	G = Bit-plane 4-1 contents

COLOR SELECTION IN HIGH-RESOLUTION MODE

Table 3-14 shows playfield color selection in high-resolution mode. If the bit-combinations from the playfields are as shown, the color is taken from the color register number indicated.

Table 3-14: High-resolution Color Selection

Single Playfield Bit-planes 4,3,2,1	Dual Playfields	Color Register Number
	Playfield 1 Bit-planes 3,1	
0000	00 *	0 **
0001	01	1
0010	10	2
0011	11	3
0100	NOT USED IN THIS MODE	4
0101		5
0110		6
0111		7
	Playfield 2 Bit-planes 4,2	
1000	00 *	8
1001	01	9
1010	10	10
1011	11	11
1100	NOT USED IN THIS MODE	12
1101		13
1110		14
1111		15

---

* Selects "transparent" mode.

** Color register 0 always defines the background color.

Chapter 4

SPRITE HARDWARE

Introduction

Sprites are hardware objects that are created and moved independently of the playfield display and independently of each other. Together with playfields, sprites form the graphics display of the Amiga. You can create more complex animation effects by using the blitter, which is described in the chapter called "Blitter Hardware." Sprites are produced on-screen by eight special-purpose sprite DMA channels. Basic sprites are 16 pixels wide and any number of lines high. You can choose from three colors for a sprite's pixels, and a pixel may also be transparent, showing any object behind the sprite. For larger or more complex objects, or for more color choices, you can combine sprites.

Sprite DMA channels can be reused several times within the same display field. Thus, you are not limited to having only eight sprites on the screen at the same time.

ABOUT THIS CHAPTER

This chapter discusses the following topics:

- Defining the size, shape, color, and screen position of sprites.
- Displaying and moving sprites.
- Combining sprites for more complex images, additional width, or additional colors.
- Reusing a sprite DMA channel multiple times within a display field to create more than eight sprites on the screen at one time.

Forming a Sprite

To form a sprite, you must first define it and then create a formal data structure in memory. You define a sprite by specifying its characteristics:

- On-screen width of up to 16 pixels.
- Unlimited height.
- Any shape.
- A combination of three colors, plus transparent.
- Any position on the screen.

SCREEN POSITION

A sprite's screen position is defined as a set of X,Y coordinates. Position (0,0), where X = 0 and Y = 0, is the upper left-hand corner of the display. You define a sprite's location by specifying the coordinates of its upper left-hand pixel. Sprite position is always defined as though the display modes were low-resolution and non-interlaced. The X,Y coordinate system and definition of a sprite's position are graphically represented in figure 4-1. Notice that because of display overscan, position (0,0) (that is, X = 0, Y = 0) is not normally in a viewable region of the screen.

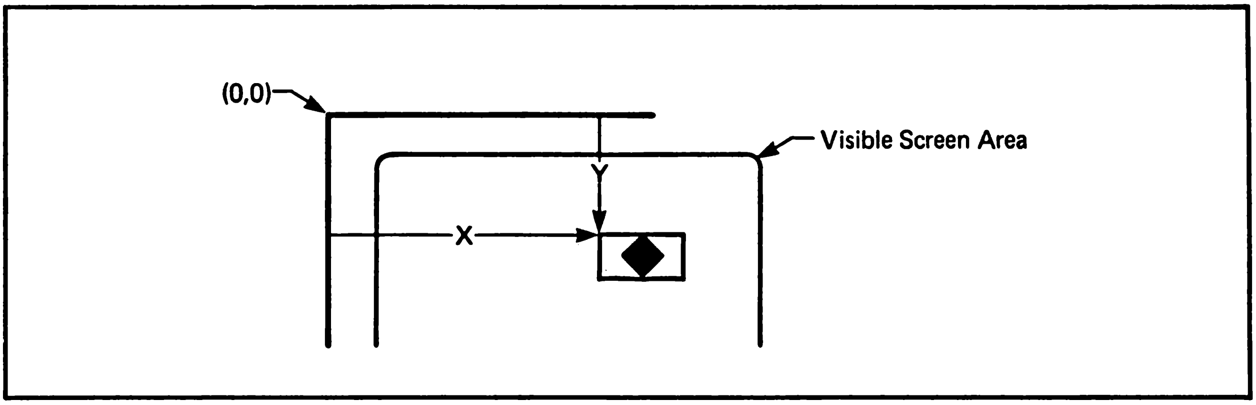


Figure 4-1: Defining Sprite On-screen Position

The amount of viewable area is also affected by the size of the playfield display window. See the "Playfield Hardware" chapter for more information about overscan and display windows.

Horizontal Position

A sprite's horizontal position (X value) can be at any pixel on the screen from 0 to 447. To be visible, however, an object must be within the boundaries of the playfield display window. In addition, the normally usable range of the video screen is from pixel 64 to pixel 383 (that is, 320 pixels of usable width). A larger area is actually scanned by the video beam but is not usually visible on the screen.

If you specify an X value for the sprite of less than 64 or an X value outside the display window, part or all of the sprite may not appear on the screen. This is sometimes desirable; such a sprite is said to be "clipped."

To make a sprite appear, unclipped, in its correct on-screen horizontal position, add 64 to the X value. For example, to make the upper leftmost pixel of a sprite appear at a position 94 pixels from the left edge of the screen, you would perform this calculation:

Desired X position	94
32 off-screen lines	+64
	158

Thus, 158 becomes the X value, which will be written into the data structure.

Note that the X position represents the location of the *very first* (leftmost) pixel in the full 16-bit-wide sprite. This is always the case, even if the leftmost pixels are specified as transparent and do not appear on the screen. If the sprite shown in figure 4-2 were located at an X value of 158, the actual image would begin on-screen four pixels later at 162. The first four pixels in this sprite are transparent and allow the background to show through.

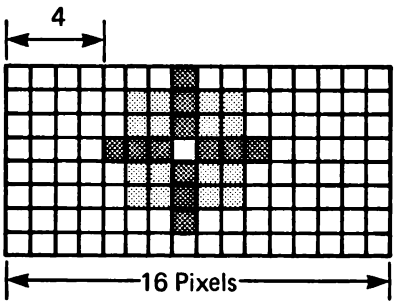


Figure 4-2: Position of Sprites

Vertical Position

You can select any position from line 0 to line 262 for the topmost edge of the sprite. The normal usable range of the video screen, however, is from line 44 through line 243. This allows the normal display height of 200 lines in non-interlaced mode. If you specify a vertical position (Y value) of less than 44, the top edge of the sprite may not appear on screen.

To make a sprite appear in its correct on-screen vertical position, add 44 to the desired position. For example, to make the upper leftmost pixel appear 25 lines below the top edge of the screen, perform this calculation:

Desired Y position	25
44 above-screen lines	+44
	69

Thus, 69 is the Y value you will write into the data structure.

Clipped Sprites

As noted above, sprites will be partially or totally clipped if they pass across or beyond the boundaries of the display window. The values of 64 (horizontal) and 44 (vertical) are "normal" for a centered display on a standard video monitor. If you choose other values to establish your display window, your sprites will be clipped accordingly.

SIZE OF SPRITES

Sprites are 16 pixels wide and can be almost any height you wish — as short as one line or taller than the screen. You would probably move a very tall sprite vertically to display a portion of it at a time.

Sprite size is based on a pixel that is 1/320th of a normal screen's width and 1/200th of a normal screen's height. This pixel size corresponds to the low-resolution and non-interlaced modes of the normal full-size playfield. Sprites, however, are independent of playfield modes of display, so changing the resolution or interlace mode of the playfield has *no effect* on the size or resolution of a sprite.

SHAPE OF SPRITES

A sprite can have any shape that will fit within the 16-pixel width. You define a sprite's shape by specifying which pixels actually appear in each of the sprite's locations. For example, figures 4-3 and 4-4 show a spaceship whose shape is marked by Xs. The first figure shows only the spaceship as you might sketch it out on graph paper. The second figure shows the spaceship within the 16-pixel width. The 0s around the spaceship mark the part of the sprite not covered by the spaceship and transparent when displayed.

```
        X X
    X X X X X X
X X X X X X X X X X
X X X X X X X X X X
    X X X X X X
        X X
```

Figure 4-3: Shape of Spaceship

```
0 0 0 0 X X 0 0 0 0 0 0 0 0 0 0
0 0 X X X X X X 0 0 0 0 0 0 0 0
X X X X X X X X X X 0 0 0 0 0 0
X X X X X X X X X X 0 0 0 0 0 0
0 0 0 X X X X 0 0 0 0 0 0 0 0 0
```

Figure 4-4: Sprite with Spaceship Shape Defined

In this example, the widest part of the shape is ten pixels and the shape is shifted to the left of the sprite. Whenever the shape is narrower than the sprite, you can control which part of the sprite is used to define the shape. This particular shape could also start at any of the pixels from 2-7 instead of pixel 1.

SPRITE COLOR

When sprites are used individually (that is, not "attached" as described under "Attached Sprites" later), each pixel can be one of three colors or transparent. Colors are selected in much the same manner as playfield colors. Figure 4-5 shows how the color of each pixel in a sprite is determined.

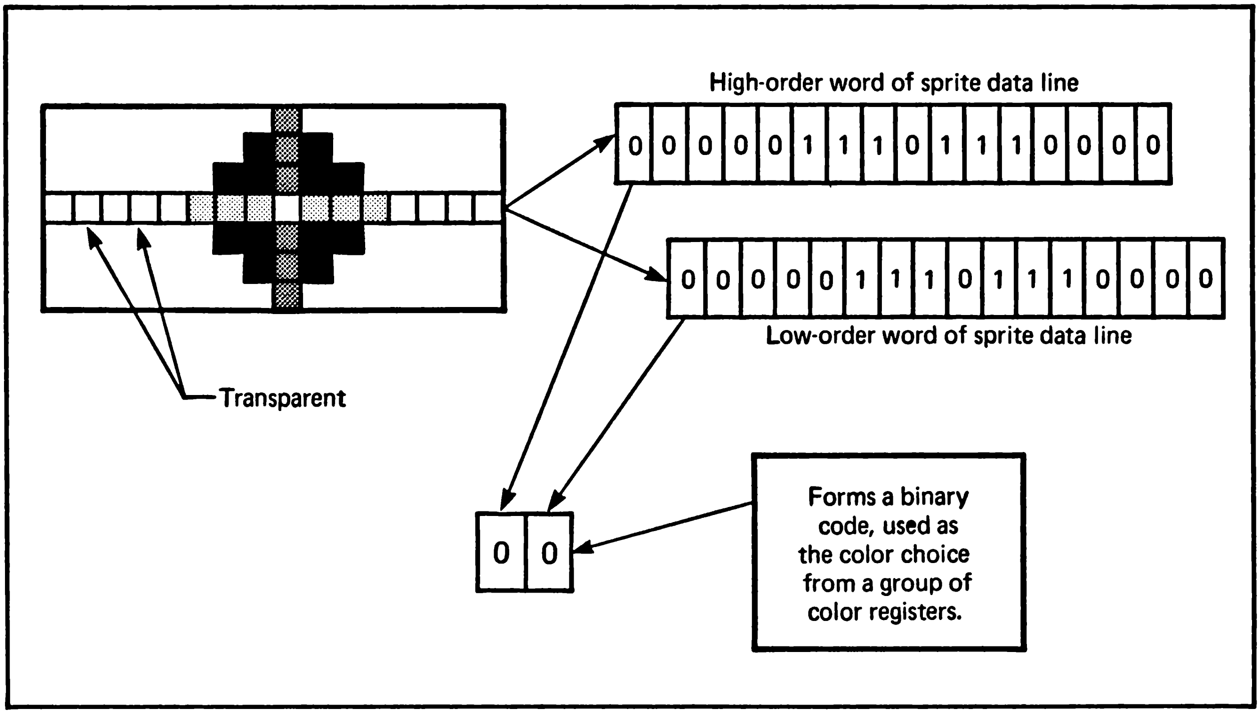

High-order word of sprite data line
0 0 0 0 0 1 1 1 0 1 1 1 0 0 0 0
0 0 0 0 0 1 1 1 0 1 1 1 0 0 0 0
Low-order word of sprite data line
Transparent
0 0
Forms a binary code, used as the color choice from a group of color registers.

Figure 4-5: Sprite Color Definition

The 0s and 1s in the two data words that define each line of a sprite in the data structure form a binary number. This binary number points to one of the four color registers assigned to that particular sprite DMA channel. The eight sprites use system color registers 16 - 31. For purposes of color selection, the eight sprites are organized into pairs and each pair uses four of the color registers as shown in figure 4-6. Note that the color value of the first register in each group of four registers is ignored by sprites. When the sprite bits select this register, the "transparent" value is used.

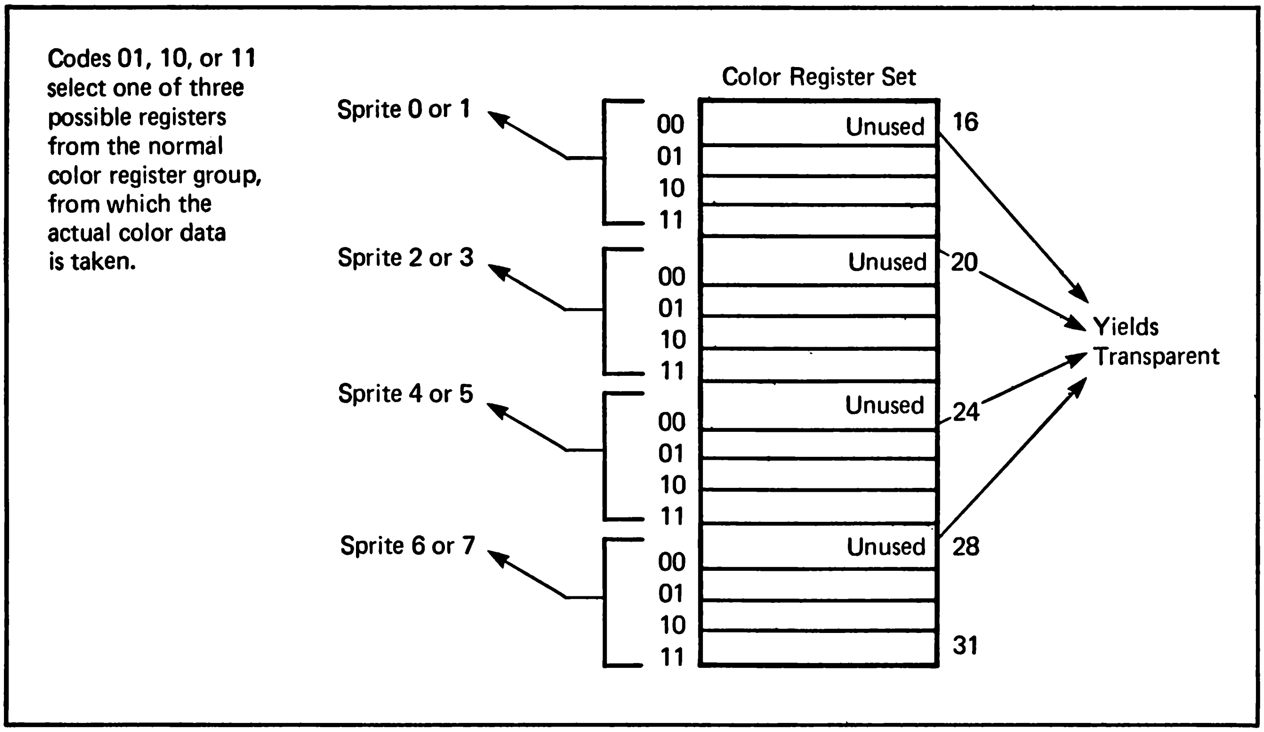

Codes 01, 10, or 11 select one of three possible registers from the normal color register group, from which the actual color data is taken.
Sprite 0 or 1
Sprite 2 or 3
Sprite 4 or 5
Sprite 6 or 7
00
01
10
11
Color Register Set
Unused
16
20
24
28
31
Yields Transparent

Figure 4-6: Color Register Assignments

If you require certain colors in a sprite, you will want to load the sprite's color registers with those colors. The "Playfield Hardware" chapter contains instructions on loading color registers.

The binary number 00 is special in this color scheme. A pixel whose value is 00 becomes transparent and shows the color of any other sprite or playfield that has lower video priority. An object with low priority appears "behind" an object with higher priority. Each sprite has a fixed video priority with respect to all the other sprites. You can vary the priority between sprites and playfields. (See chapter 7, "System Control Hardware," for more information about sprite priority.)

DESIGNING A SPRITE

For design purposes, it is convenient to lay out the sprite on paper first. You can show the desired colors as numbers from 0 to 3. For example, the spaceship shown above might look like this:

```
0000122332210000
0001223333221000
0012223333222100
0001223333221000
0000122332210000
```

The next step is to convert the numbers 0-3 into binary numbers, which will be used to build the color descriptor words of the sprite data structure. The section below shows how to do this.

BUILDING THE DATA STRUCTURE

After defining the sprite, you need to build its data structure, which is a series of 16-bit words in a contiguous memory area. Some of the words contain position and control information and some contain color descriptions. To create a sprite's data structure, you need to:

- Write the horizontal and vertical position of the sprite into the first control word.
- Write the vertical stopping position into the second control word.
- Translate the decimal color numbers 0 - 3 in your sprite grid picture into binary color numbers. Use the binary values to build color descriptor (data) words and write these words into the data structure.
- Write the control words that indicate the end of the sprite data structure.

Table 4-1 shows a sprite data structure with the memory location and function of each word:

Table 4-1: Sprite Data Structure

Memory Location	**16-bit Word**	**Function**
N	Sprite control word 1	Vertical and horizontal start position
N+1	Sprite control word 2	Vertical stop position
N+2	Color descriptor low word	Color bits for line 1
N+3	Color descriptor high word	Color bits for line 1
N+4	Color descriptor low word	Color bits for line 2
N+5	Color descriptor high word	Color bits for line 2
	.	
	.	
	.	
	End-of-data words	Two words indicating the next usage of this sprite

All memory addresses for sprites are word addresses. You will need enough contiguous memory to provide room for two words for the control information, two words for each horizontal line in the sprite, and two end-of-data words.

Because this data structure must be accessible by the special-purpose chips, you must ensure that this data is located within the lowest 512K bytes of the system memory.

Figure 4-7 shows how the data structure relates to the sprite.

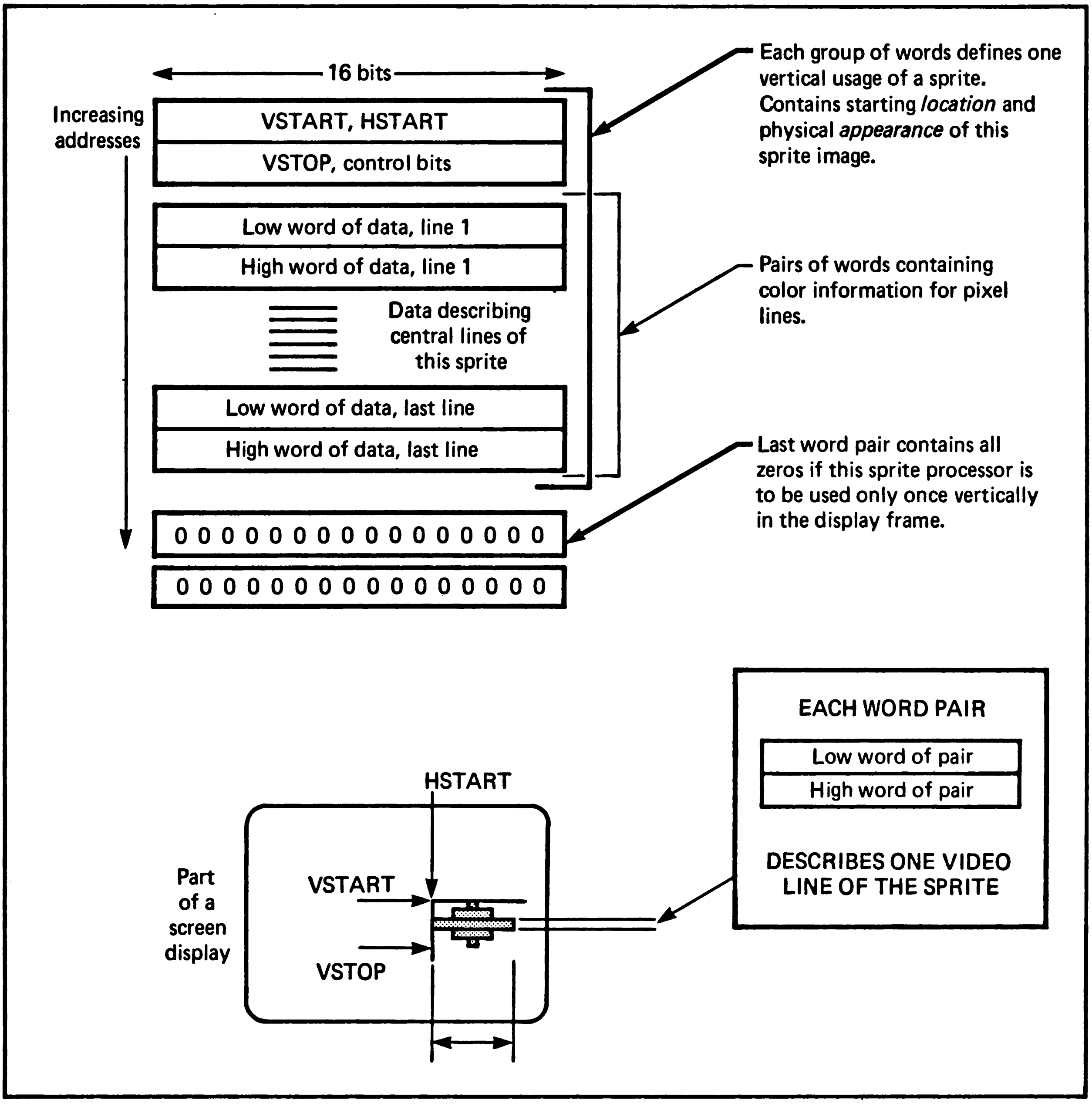

16 bits
Increasing addresses
VSTART, HSTART
VSTOP, control bits
Low word of data, line 1
High word of data, line 1
Data describing central lines of this sprite
Low word of data, last line
High word of data, last line
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Each group of words defines one vertical usage of a sprite. Contains starting *location* and physical *appearance* of this sprite image.
Pairs of words containing color information for pixel lines.
Last word pair contains all zeros if this sprite processor is to be used only once vertically in the display frame.
HSTART
Part of a screen display
VSTART
VSTOP
EACH WORD PAIR
Low word of pair
High word of pair
DESCRIBES ONE VIDEO LINE OF THE SPRITE

Figure 4-7: Data Structure Layout

Sprite Control Word 1 : SPRxPOS

This word contains the vertical (VSTART) and horizontal (HSTART) starting position for the sprite. This is where the topmost line of the sprite will be positioned.

Bits 15-8 contain the low 8 bits of VSTART
Bits 7-0 contain the high 8 bits of HSTART

Sprite Control Word 2 : SPRxCTL

This word contains the vertical stopping position of the sprite on the screen. It also contains some data having to do with sprite attachment, which is described later on.

SPRxCTL

Bits 15-8	The low eight bits of VSTOP
Bit 7	(Used in attachment)
Bits 6-3	Unused (make zero)
Bit 2	The VSTART high bit
Bit 1	The VSTOP high bit
Bit 0	The HSTART low bit

The value (VSTOP - VSTART + 1) defines how many lines high the sprite will be.

Sprite Color Descriptor Words

It takes two color descriptor words to describe each horizontal line of a sprite; the high-order word and the low-order word. To calculate how many color descriptor words you need, multiply the height of the sprite in lines by 2. The bits in the high-order color descriptor word contribute the leftmost digit of the binary color selector number for each pixel; the low-order word contributes the rightmost digit.

To form the color descriptor words, you first need to form a picture of the sprite, showing the color of each pixel as a number from 0 - 3. Each number represents one of the colors in the sprite's color registers. For example, here is the spaceship sprite again:

```
0000122332210000
0001223333221000
0012223333222100
0001223333221000
0000122332210000
```

Next, you translate each of the numbers in this picture into a binary number. The first line in binary is shown below. The binary numbers are represented vertically with the low digit in the top line and the high digit right below it. This is how the two color descriptor words for each sprite line are written in memory.

```
0 0 0 0 1 0 0 1 1 0 0 1 0 0 0 0
0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0
```

The first line above becomes the color descriptor high word for line 1 of the sprite. The second line becomes the color descriptor low word. In this fashion, you translate each line in the sprite into binary 0s and 1s.

Each of the binary numbers formed by the combination of the two data words for each line refers to a specific color register in that particular sprite channel's segment of the color table. Sprite channel 0, for example, takes its colors from registers 17 - 19. The binary numbers corresponding to the color registers for sprite DMA channel 0 are shown in table 4-2.

Table 4-2: Sprite Color Registers

Binary Number	Color Register Number
00	Transparent
01	17
10	18
11	19

Recall that binary 00 always means transparent and never refers to a color.

End-of-data Words

When the vertical position of the beam counter is equal to the VSTOP value in the sprite control words, the next two words fetched from the sprite data structure are written into the sprite control registers instead of being sent to the color registers. These two words are interpreted by the hardware in the same manner as the original words that were first loaded into the control registers. If the VSTART value contained in these words is lower than the current beam position, this sprite will not be reused in this display field. For consistency, the value 0 should be used for both words when ending the usage of a sprite. Sprite reuse is discussed later.

The following data structure is for the spaceship sprite. It will be located at V = 65 and H = 128 on the screen.

```
SPRITE:
        DC.W    $6D60,$7200     ;VSTART, HSTART, VSTOP
        DC.W    $0990,$07E0     ;First pair of descriptor words
        DC.W    $13C8,$0FF0
        DC.W    $23C4,$1FF8
        DC.W    $13C8,$0FF0
        DC.W    $0990,$07E0
        DC.W    $0000,$0000     ;End of sprite data
```

Displaying a Sprite

After building the data structure, you need to tell the system to display it. This section describes the display of sprites in "automatic" mode. In this mode, once the sprite DMA channel begins to retrieve and display the data, the display continues until the VSTOP position is reached. Manual mode is described later on in this chapter.

The following steps are used in displaying the sprite:

1. Decide which of the eight sprite DMA channels to use.

2. Set the sprite pointers to tell the system where to find the sprite data.

3. Turn on sprite direct memory access if it is not already on.

4. For each subsequent display field, during the vertical blanking interval, rewrite the sprite pointers.

CAUTION

If sprite DMA is turned off while a sprite is being displayed (that is, after VSTART but before VSTOP), the system will continue to display the line of sprite data that was most recently fetched. This causes a vertical bar to appear on the screen. It is recommended that sprite DMA be turned off only during vertical blanking or during some portion of the display where you are *sure* that no sprite is being displayed.

SELECTING A DMA CHANNEL AND SETTING THE POINTERS

In deciding which DMA channel to use, you should take into consideration the colors assigned to the sprite and the sprite's video priority.

The sprite DMA channel uses two pointers to read in sprite data and control words. During the vertical blanking interval before the first display of the sprite, you need to write the sprite's memory address into these pointers. The pointers for each sprite are called SPRxPTH and SPRxPTL, where "x" is the number of the sprite DMA channel. SPRxPTH points to the high three bits of the memory address of the first word in the sprite and SPRxPTL points to the low fifteen bits. As usual, you can write a long word into SPRxPTH.

In the following example the processor initializes the data pointers for sprite 0. Normally, this is done by the Copper. The sprite is at address $20000.

```
SPR0PTH  EQU     $DFF120
SPR0PTL  EQU     $DFF122
;
         MOVE.L  #$20000,SPR0PTH  ;Write $20000 to sprite 0 pointer
```

These pointers are dynamic; they are incremented by the sprite DMA channel to point first to the control words, then to the data words, and finally to the end-of-data words. After reading in the sprite control information and storing it in other registers, they proceed to read in the color descriptor words. The color descriptor words are stored in sprite data registers, which are used by the sprite DMA channel to display the data on screen. For more information about how the sprite DMA channels handle the display, see the "Hardware Details" section below.

RESETTING THE ADDRESS POINTERS

For one single display field, the system will automatically read the data structure and produce the sprite on-screen in the colors that are specified in the sprite's color registers. If you want the sprite to be displayed in subsequent display fields, you must rewrite the contents of the sprite pointers during each vertical blanking interval. This is necessary because during the display field, the pointers are incremented to point to the data which is being fetched as the screen display progresses.

The rewrite becomes part of the vertical blanking routine, which can be handled by instructions in the Copper lists.

SPRITE DISPLAY EXAMPLE

This example displays the spaceship sprite at location V = 65, H = 128. The equates are the usual, so they're not repeated here.

```
; First, we set up a single bit-plane.
;
        LEA     CUSTOM,A0                   ;Point A0 at custom chips
        MOVE.W  #$1200,BPLCON0(A0)          ;1 bit-plane color is on
        MOVE.W  #$0000,BPL1MOD(A0)          ;Modulo = 0
        MOVE.W  #$0000,BPLCON1(A0)          ;Horizontal scroll value = 0
        MOVE.W  #$0024,BPLCON2(A0)          ;Sprites have priority over playfields
        MOVE.W  #$0038,DDFSTRT(A0)          ;Set data-fetch start
        MOVE.W  #$00D0,DDFSTOP(A0)          ;Set data-fetch stop
        MOVE.W  #$2C81,DIWSTRT(A0)          ;Set display window start
        MOVE.W  #$F4C1,DIWSTOP(A0)          ;Set display window stop
;
; Set up color registers.
;
        MOVE.W  #$0008,COLOR00(A0)          ;Background color = dark blue
        MOVE.W  #$0000,COLOR01(A0)          ;Foreground color = black
        MOVE.W  #$0FF0,COLOR17(A0)          ;Color 17 = yellow
        MOVE.W  #$00FF,COLOR18(A0)          ;Color 18 = cyan
        MOVE.W  #$0F0F,COLOR19(A0)          ;Color 19 = magenta
;
; Move Copper list to $20000.
;
        MOVE.L  #$20000,A1                  ;Point A1 at Copper list destination
        LEA     COPPERL,A2                  ;Point A2 at Copper list source
```

```
CLOOP:
        MOVE.L  (A2),(A1)+                  ;Move a long word
        CMP.L   #$FFFFFFFE,(A2)+            ;Check for end of list
        BNE     CLOOP                       ;Loop until entire list is moved
;
; Move sprite to $25000.
;
        MOVE.L  #$25000,A1                  ;Point A1 at sprite destination
        LEA     SPRITE,A2                   ;Point A2 at sprite source
SPRLOOP:
        MOVE.L  (A2),(A1)+                  ;Move a long word
        CMP.L   #$00000000,(A2)+            ;Check for end of sprite
        BNE     SPRLOOP                     ;Loop until entire sprite is moved
;
; Now we write a dummy sprite to $30000, since all eight sprites are activated
; at the same time and we're only going to use one.  The remaining sprites
; will point to this dummy sprite data.
;
        MOVE.L  #$00000000,$30000           ;Write it
;
; Point Copper at Copper list.
;
        MOVE.L  #$20000,CUSTOM+COP1LC
;
; Fill bit-plane with $FFFFFFFF.
;
        MOVE.L  #$21000,A1                  ;Point A1 at bit-plane
        MOVE.W  #2000,D0                    ;2000 long words = 8000 bytes
FLOOP:
        MOVE.L  #$FFFFFFFF,(A1)+            ;Move a long word of $FFFFFFFF
        SUBQ.W  #1,D0                       ;Decrement counter
        BNE     FLOOP                       ;Loop until bit-plane is full
;
; Start DMA.
;
        MOVE.W  CUSTOM+COPJMP1,D0           ;Force load into Copper
                                            ;  program counter
        MOVE.W  #$83A0,(CUSTOM+DMACON)      ;Bit-plane, Copper, and sprite DMA
        BRA                                 ;....next things to do...

;
; This is a Copper list for one bit-plane, and 8 sprites.  The bit-plane lives
; at $21000.  Sprite 0 lives at $25000; all others live at $30000 (the dummy sprite).
```

```
;
COPPERL:
        DC.W    $00E0,$0002             ;Bit plane 1 pointer = $21000
        DC.W    $00E2,$1000
        DC.W    $0120,$0002             ;Sprite 0 pointer = $25000
        DC.W    $0122,$5000
        DC.W    $0124,$0003             ;Sprite 1 pointer = $30000
        DC.W    $0126,$0000
        DC.W    $0128,$0003             ;Sprite 2 pointer = $30000
        DC.W    $012A,$0000
        DC.W    $012C,$0003             ;Sprite 3 pointer = $30000
        DC.W    $012E,$0000
        DC.W    $0130,$0003             ;Sprite 4 pointer = $30000
        DC.W    $0132,$0000
        DC.W    $0134,$0003             ;Sprite 5 pointer = $30000
        DC.W    $0136,$0000
        DC.W    $0138,$0003             ;Sprite 6 pointer = $30000
        DC.W    $013A,$0000
        DC.W    $013C,$0003             ;Sprite 7 pointer = $30000
        DC.W    $013E,$0000
        DC.W    $FFFF,$FFFE             ;End of Copper list
;
; Sprite data for spaceship sprite.  It appears on the screen at V=65 and H=128.
;
SPRITE:
        DC.W    $6D60,$7200             ;VSTART, HSTART, VSTOP
        DC.W    $0990,$07E0             ;First pair of descriptor words
        DC.W    $13C8,$0FF0
        DC.W    $23C4,$1FF8
        DC.W    $13C8,$0FF0
        DC.W    $0990,$07E0
        DC.W    $0000,$0000             ;End of sprite data
```

Moving a Sprite

A sprite generated in automatic mode can be moved by specifying a different position in the data structure. For each display field, the data is reread and the sprite redrawn. Therefore, if you change the position data before the sprite is redrawn, it will appear in a new position and will seem to be moving.

You must take care that you are not moving the sprite (that is, changing control word data) at the same time that the system is using that data to find out where to display the object. If you do so, the system might find the start position for one field and the stop position for the following field as it retrieves data for display. This would cause a "glitch" and would mess up the screen. Therefore, you should change the content of the control words only during a time when the system is not trying to read them. Usually, the vertical blanking period is a safe time, so moving the sprites becomes part of the vertical blanking tasks and is handled by the Copper as shown in the example below.

As sprites move about on the screen, they can collide with each other or with either of the two playfields. You can use the hardware to detect these collisions and exploit this capability for special effects. In addition, you can use collision detection to keep a moving object within specified on-screen boundaries. Collision is described in chapter 7, "System Control Hardware."

In this example of moving a sprite, the spaceship is bounced around on the screen, changing direction whenever it reaches an edge.

The sprite position data, containing VSTART and HSTART, lives in memory at $25000. VSTOP is located at $25002. You write to these locations to move the sprite. Once during each frame, VSTART is incremented (or decremented) by 1 and HSTART by 2. Then a new VSTOP is calculated, which will be the new VSTART + 6.

```
        MOVE.B  #151,D0                     ;Initialize horizontal count
        MOVE.B  #194,D1                     ;Initialize vertical count
        MOVE.B  #64,D2                      ;Initialize horizontal position
        MOVE.B  #44,D3                      ;Initialize vertical position
        MOVE.B  #1,D4                       ;Initialize horizontal increment value
        MOVE.B  #1,D5                       ;Initialize vertical increment value
;
;Here we wait for the vertical blanking bit in INTREQR to turn on.
; This ensures a glitch-free display.
;
VLOOP:
        MOVE.W  CUSTOM+INTREQR,D6           ;Read interrupt request word
        AND.W   #$0020,D6                   ;Mask off all but vertical blank bit
        BEQ     VLOOP                       ;Loop until bit is a 1
        MOVE.W  #$0020,CUSTOM+INTREQ        ;Vertical bit is on, so reset it
;
        ADD.B   D4,D2               ;Increment horizontal value
        SUBQ.B  #,D0                ;Decrement horizontal counter
        BNE     L1
        MOVE.B  #151,D0             ;Count exhausted, reset to 151
        EOR.B   #$FE,D4             ;Negate the increment value
```

```
L1:     MOVE.B  D2,$25001       ;Write new HSTART value to sprite
        ADD.B   D5,D3           ;Increment vertical value
        SUBQ.B  #1,D1           ;Decrement vertical counter
        BNE     L2
        MOVE.B  #194,D1         ;Count exhausted, reset to 194
        EOR.B   #$FE,D5         ;Negate the increment value
L2:     MOVE.B  D3,$25000       ;Write new VSTART value to sprite
        MOVE.B  D3,D6           ;Must now calculate new VSTOP
        ADD.B   #6,D6           ;VSTOP always VSTART+6 for spaceship
        MOVE.B  D6,$25002       ;Write new VSTOP to sprite
        BRA     VLOOP           ;Loop forever
```

Creating Additional Sprites

To use additional sprites, you must create a data structure for each one and arrange the display as shown in the previous section, naming the pointers SPR1PTH and SPR1PTL for sprite DMA channel 1, SPR2PTH and SPR2PTL for sprite DMA channel 2, and so on.

Note that when you enable sprite DMA for one sprite, you enable DMA for all the sprites and place them all in automatic mode. Thus, you do not need to repeat this step when using additional sprite DMA channels. Once the sprite DMA channels are enabled, all eight sprite pointers *must* be initialized to either a real sprite or a safe null sprite. An uninitialized sprite could cause spurious sprite video to appear.

Also, recall that each pair of sprites takes its color from different color registers, as shown in table 4-3.

Table 4-3: Color Registers for Sprite Pairs

Sprite Numbers	Color Registers
0 and 1	17 - 19
2 and 3	21 - 23
4 and 5	25 - 27
6 and 7	29 - 31

When you have more than one sprite on the screen, you may need to take into consideration their relative video priority, that is, which sprite appears in front of or behind another. Each sprite has a fixed video priority with respect to all the others. The lowest numbered sprite has the highest priority and appears in front of all other sprites; the highest numbered sprite has the lowest priority. This is illustrated in figure 4-8.

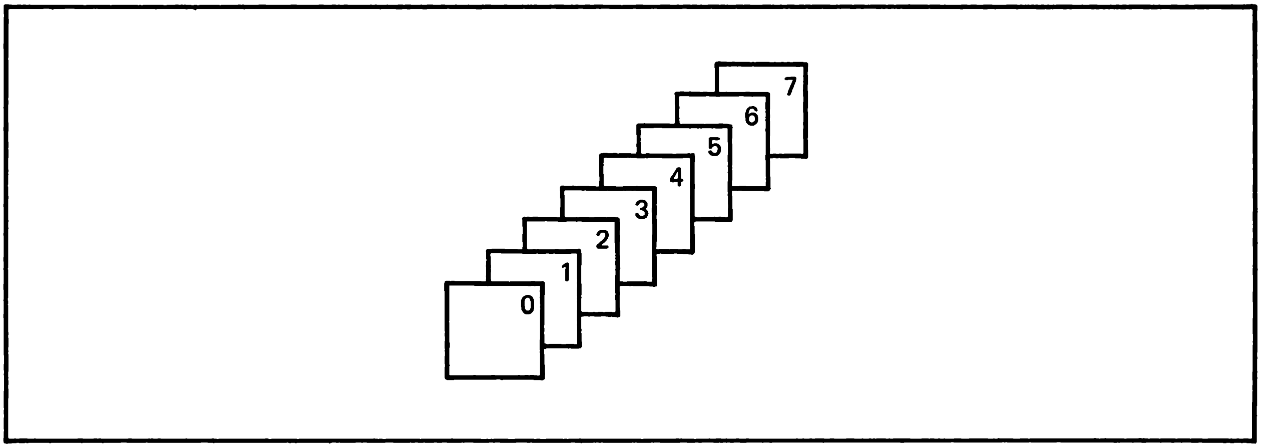
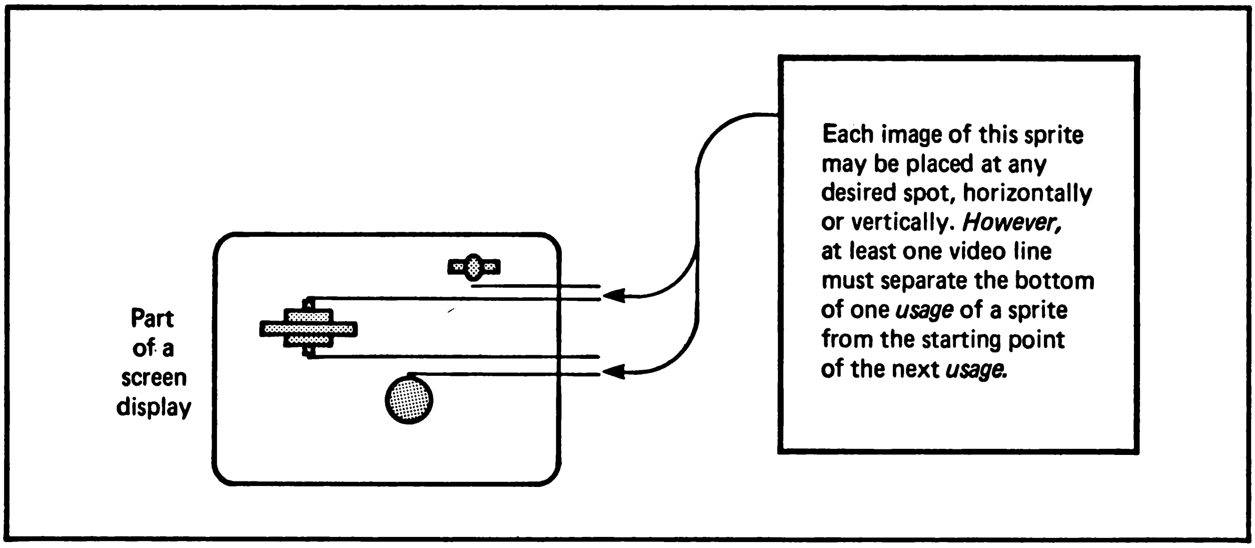


Figure 4-8: Sprite Priority

Reusing Sprite DMA Channels

Each of the eight sprite DMA channels can produce more than one independently controllable image. There may be times when you want more than eight objects, or you may be left with fewer than eight objects because you have attached some of the sprites to produce more colors or larger objects or overlapped some to produce more complex images. You can reuse each sprite DMA channel several times within the same display field, as shown in figure 4-9.



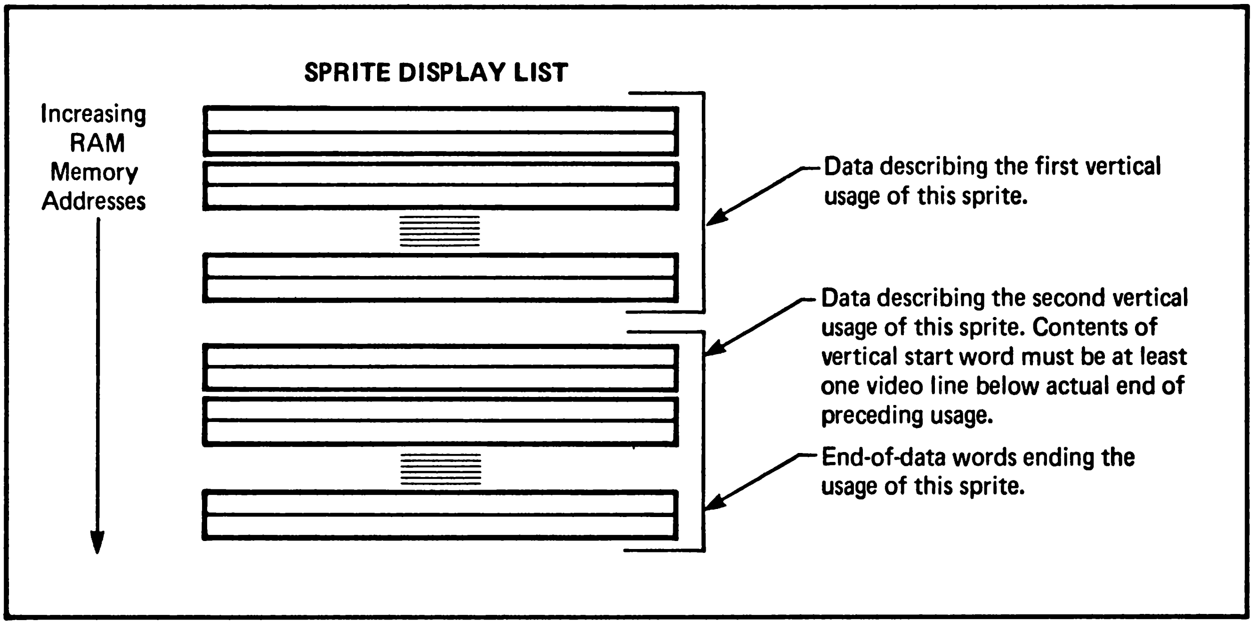
Each image of this sprite may be placed at any desired spot, horizontally or vertically. *However,* at least one video line must separate the bottom of one *usage* of a sprite from the starting point of the next *usage.*

Figure 4-9: Typical Example of Sprite Reuse

In single-sprite usage, two all-zero words are placed at the end of the data structure to stop the DMA channel from retrieving any more data for that particular sprite during that display field. To reuse a DMA channel, you replace this pair of zero words with another complete sprite data structure, which describes the reuse of the DMA channel at a position lower on the screen than the first use. You place the two all-zero words at the end of the data structure that contains the information for all usages of the DMA channel. For example, figure 4-10 shows the data structure that describes the picture above.

The only restrictions on the reuse of sprites during a single display field is that the bottom line of one usage of a sprite must be separated from the top line of the next usage by at least one horizontal scan line. This restriction is necessary because only two DMA cycles per horizontal scan line are allotted to each of the eight channels. The sprite channel needs the time during the blank line to fetch the control word describing the next usage of the sprite.

The following example displays the spaceship sprite and then redisplays it as a different object. Only the sprite data list is affected, so only the data list is shown here. However, the sprite looks best with the color registers set as shown in the example.



Data describing the first vertical usage of this sprite.

Data describing the second vertical usage of this sprite. Contents of vertical start word must be at least one video line below actual end of preceding usage.

End-of-data words ending the usage of this sprite.

Figure 4-10: Typical Data Structure for Sprite Re-use

```
        LEA     CUSTOM,A0
        MOVE.W  #$0F00,COLOR17(A0)  ;Color 17 = red
        MOVE.W  #$0FF0,COLOR18(A0)  ;Color 18 = yellow
        MOVE.W  #$0FFF,COLOR19(A0)  ;Color 19 = white

SPRITE:

        DC.W    $6D60,$7200
        DC.W    $0990,$07E0
        DC.W    $13C8,$0FF0
        DC.W    $23C4,$1FF8
        DC.W    $13C8,$0FF0
        DC.W    $0990,$07E0
        DC.W    $8080,$8D00                 ;VSTART, HSTART, VSTOP for new sprite
        DC.W    $1818,$0000
        DC.W    $7E7E,$0000
        DC.W    $7FFE,$0000
        DC.W    $FFFF,$2000
        DC.W    $FFFF,$2000
        DC.W    $FFFF,$3000
        DC.W    $FFFF,$3000
        DC.W    $7FFE,$1800
        DC.W    $7FFE,$0C00
DC.W    $3FFC,$0000
DC.W    $0FF0,$0000
DC.W    $03C0,$0000
DC.W    $0180,$0000
DC.W    $0000,$0000              ;End of sprite data
```

Overlapped Sprites

For more complex or larger moving objects, you can overlap sprites. Overlapping simply means that the sprites have the same or relatively close screen positions. A relatively close screen position can result in an object that is wider than 16 pixels.

The built-in sprite video priority ensures that one sprite appears to be behind the other when sprites are overlapped. The priority circuitry gives the lowest-numbered sprite the highest priority and the highest numbered sprite the lowest priority. Therefore, when designing displays with overlapped sprites, make sure the "foreground" sprite has a lower number than the "background" sprite. In figure 4-11, for example, the cage should be generated by a lower-numbered sprite DMA channel than the monkey.

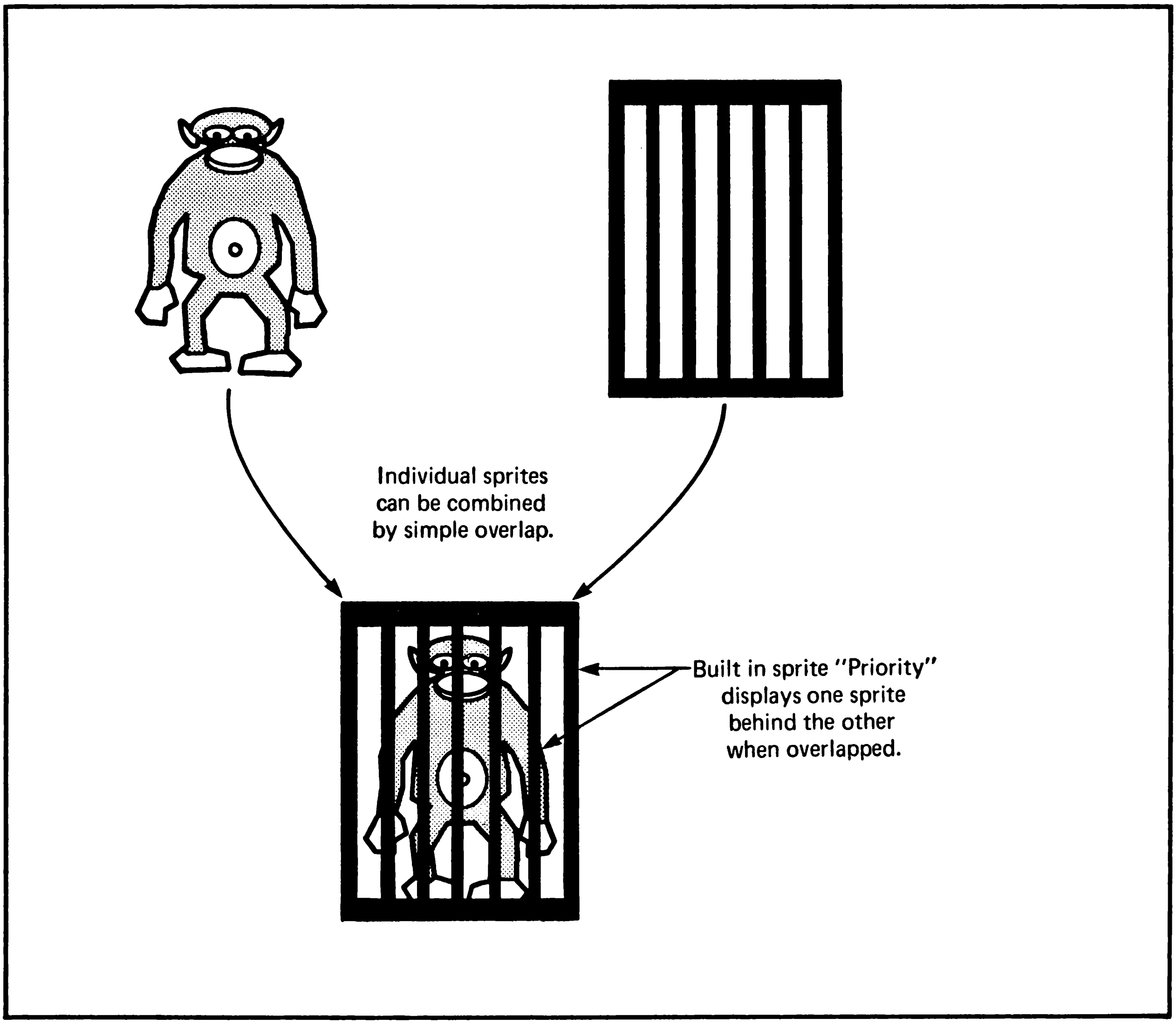

Individual sprites can be combined by simple overlap.
Built in sprite "Priority" displays one sprite behind the other when overlapped.

Figure 4-11: Overlapping Sprites (Not Attached)

You can create a wider sprite display by placing two sprites next to each other. For instance, figure 4-12 shows the spaceship sprite and how it can be made twice as large by using two sprites placed next to each other.

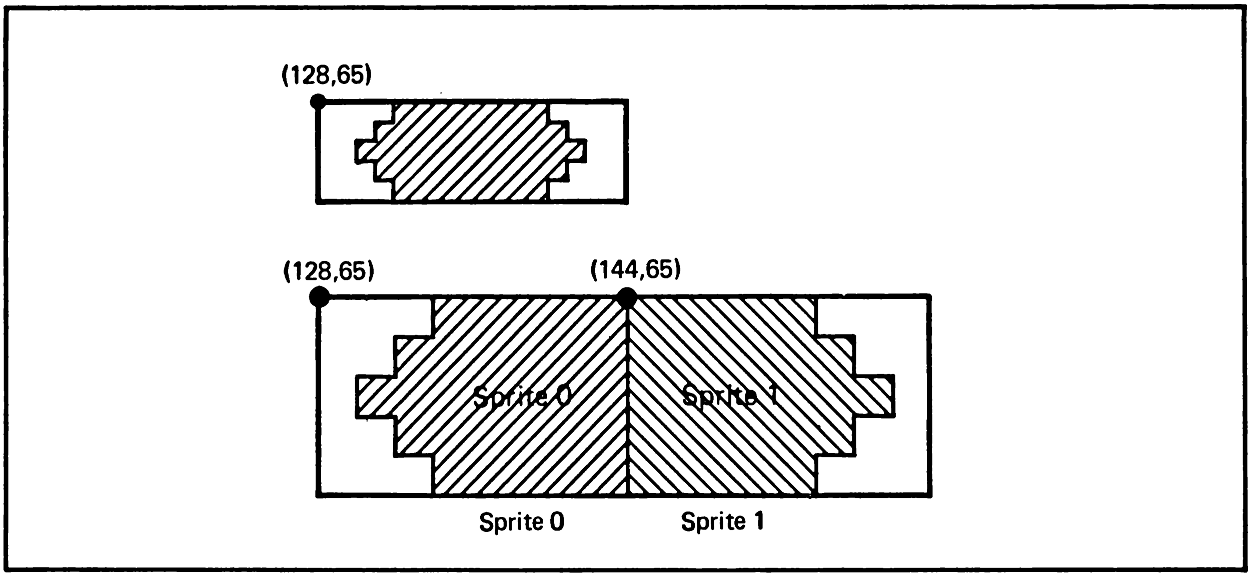


Figure 4-12: Placing Sprites Next to Each Other

Attached Sprites

You can create sprites that have fifteen possible color choices (plus transparent) instead of three (plus transparent), by "attaching" two sprites. To create attached sprites, you must:

- Use two channels per sprite, creating two sprites of the same size and located at the same position.
- Set a bit called ATTACH in the second sprite control word.

The fifteen colors are selected from the full range of color registers available to sprites — registers 17 through 31. The extra color choices are possible because each pixel contains four bits instead of only two as in the normal, unattached sprite. Each sprite in the attached pair contributes two bits to the binary color selector number. For example, if you are using sprite DMA channels 0 and 1, the high- and low-order color descriptor words for line 1 in both data structures are combined into line 1 of the attached object.

Sprites can be attached in the following combinations:

Sprite 1 to sprite 0
Sprite 3 to sprite 2
Sprite 5 to sprite 4
Sprite 7 to sprite 6

Any or all of these attachments can be active during the same display field. As an example, assume that you wish to have more colors in the spaceship sprite and you are using sprite DMA channels 0 and 1. There are five colors plus transparent in this sprite.

0000154444510000
0001564444651000
0015676446765100
0001564444651000
0000154444510000

The first line in this sprite requires the four data words shown in table 4-4 to form the correct binary color selector numbers.

Table 4-4: Data Words for First Line of Spaceship Sprite

Pixel Number

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Line 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Line 2	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
Line 3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Line 4	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0

The binary numbers 0 through 15 select registers 17 through 31 as shown in table 4-5.

Table 4-5: Color Registers in Attached Sprites

Decimal Number	**Binary Number**	**Color Register Number**
0	0000	16 *
1	0001	17
2	0010	18
3	0011	19
4	0100	20
5	0101	21
6	0110	22
7	0111	23
8	1000	24
9	1001	25
10	1010	26
11	1011	27
12	1100	28
13	1101	29
14	1110	30
15	1111	31

* Unused; yields transparent pixel.

The highest numbered sprite (number 1, in this example) contributes the highest order bits (leftmost) in the binary number. The high-order data word in each sprite contributes the leftmost digit. Therefore, the lines above are written to the sprite data structures as follows:

Line 1	Sprite 1 high-order word for sprite line 1
Line 2	Sprite 1 low-order word for sprite line 1
Line 3	Sprite 0 high-order word for sprite line 1
Line 4	Sprite 0 low-order word for sprite line 1

Attachment is in effect only when the ATTACH bit, bit 7 in sprite control word 2, is set to 1 in the data structure for the odd-numbered sprite. So, in this example, you set bit 7 in sprite control word 2 in the data structure for sprite 1.

When the sprites are moved, the Copper list must keep them both at exactly the same position relative to each other. If they are not kept together on the screen, their pixels will change color. Each sprite will revert to three colors plus transparent, but the colors may be different than if they were ordinary, unattached sprites. The color selection for the lower numbered sprite will be from color registers 17-19. The color selection for the higher numbered sprite will be from color registers 20, 24, and 28.

The following data structure is for the six-color spaceship made with two attached sprites.

```
SPRITE0:

        DC.W    $6D60,$7200         ;VSTART = 65, HSTART = 128
        DC.W    $0C30,$0000         ;First color descriptor word
        DC.W    $1818,$0420
        DC.W    $342C,$0E70
        DC.W    $1818,$0420
        DC.W    $0C30,$0000
        DC.W    $0000,$0000         ;End of sprite 0

SPRITE1:

        DC.W    $6D60,$7280         ;Same as sprite 0 except attach bit on
        DC.W    $07E0,$0000         ;First descriptor word for sprite 1
        DC.W    $0FF0,$0000
        DC.W    $1FF8,$0000
        DC.W    $0FF0,$0000
        DC.W    $07E0,$0000
        DC.W    $0000,$0000         ;End of sprite 1
```

Manual Mode

It is almost always best to load sprites using the automatic DMA channels. Sometimes, however, it is useful to load these registers directly from one of the microprocessors. Sprites may be activated "manually" whenever they are not being used by a DMA channel. The same sprite that is showing a DMA-controlled icon near the top of the screen can also be reloaded manually to show a vertical colored bar near the bottom of the screen. Sprites can be activated manually even when the sprite DMA is turned off.

You display sprites manually by writing to the sprite data registers SPRxDATB and SPRxDATA, in that order. You write to SPRxDATA last because that address "arms" the sprite to be output at the next horizontal comparison. The data written will then be displayed on every line, at the horizontal position given in the "H" portion of the position registers SPRxPOS and SPRxCTL. If the data is unchanged, the result will be a vertical bar. If the data is reloaded for every line, a complex sprite can be produced.

The sprite can be terminated ("disarmed") by writing to the SPRxCTL register. If you write to the SPRxPOS register, you can manually move the sprite horizontally at any time, even during normal sprite usage.

Sprite Hardware Details

Sprites are produced by the circuitry shown in figure 4-13. This figure shows in block form how a pair of data words becomes a set of pixels displayed on the screen.

The circuitry elements for sprite display are explained below.

- Sprite data registers. The registers SPRxDATA and SPRxDATB hold the bit patterns that describe one horizontal line of a sprite for each of the eight sprites. A line is 16 pixels wide, and each line is defined by two words to provide selection of three colors and transparent.

- Parallel-to-serial converters. Each of the 16 bits of the sprite data bit pattern is individually sent to the color select circuitry at the time that the pixel associated with that bit is being displayed on-screen.

  Immediately after the data is transferred from the sprite data registers, each parallel-to-serial converter begins shifting the bits out of the converter, most significant (leftmost) bit first. The shift occurs once during each low-resolution pixel time and continues until all 16 bits have been transferred to the display circuitry. The shifting and data output does not begin again until the next time this converter is loaded from the data registers.

  Because the video image is produced by an electron beam that is being swept from left to right on the screen, the bit-image of the data corresponds exactly to the image that actually appears on the screen (most significant data on the left).

- Sprite serial video data. Sprite data goes to the priority circuit to establish the priority between sprites and playfields.

- Sprite position registers. These registers, called SPRxPOS, contain the horizontal position value (X value) and vertical position value (Y value) for each of the eight sprites.

- Sprite control registers. These registers, called SPRxCTL, contain the stopping position for each of the eight sprites and whether or not a sprite is attached.

- Beam counter. The beam counter tells the system the current location of the video beam that is producing the picture.

- Comparator. This device compares the value of the beam counter to the Y value in the position register SPRxPOS. If the beam has reached the position at which the leftmost upper pixel of the sprite is to appear, the comparator issues a load signal to the serial-to-parallel converter and the sprite display begins.

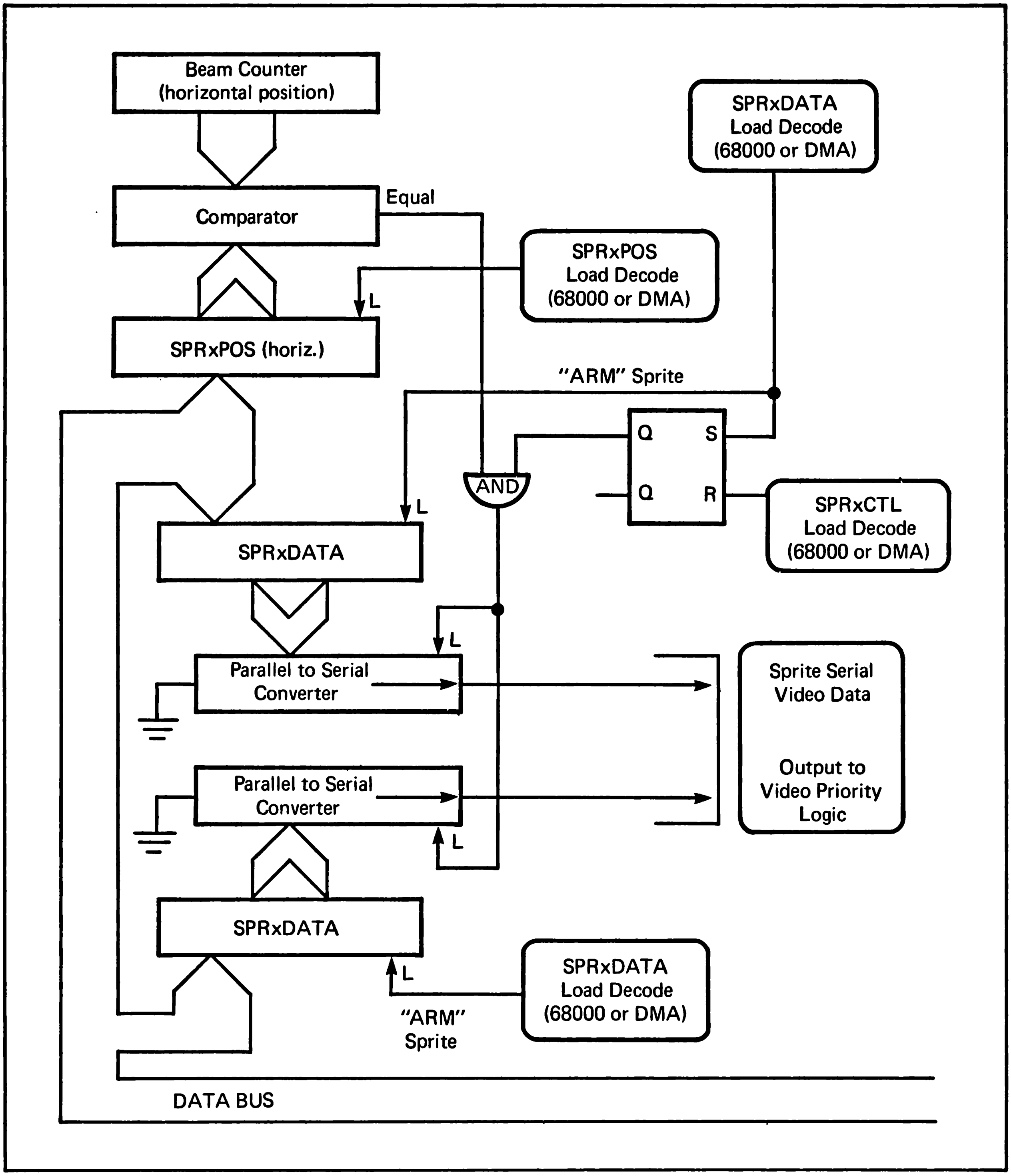

Beam Counter (horizontal position)
Comparator
Equal
SPRxDATA Load Decode (68000 or DMA)
SPRxPOS Load Decode (68000 or DMA)
L
SPRxPOS (horiz.)
"ARM" Sprite
Q
S
Q
R
AND
SPRxCTL Load Decode (68000 or DMA)
L
SPRxDATA
L
Parallel to Serial Converter
Sprite Serial Video Data
Output to Video Priority Logic
Parallel to Serial Converter
L
SPRxDATA
L
SPRxDATA Load Decode (68000 or DMA)
"ARM" Sprite
DATA BUS

Figure 4-13: Sprite Control Circuitry

Figure 4-13 shows the following:

- Writing to the sprite *control* registers *disables* the horizontal comparator circuitry. This prevents the system from sending any output from the data registers to the serial converter or to the screen.
- Writing to the sprite A *data* register *enables* the horizontal comparator. This enables output to the screen when the horizontal position of the video beam equals the horizontal value in the position register.
- If the comparator is enabled, the sprite data will be sent to the display, with the leftmost pixel of the sprite data placed at the position defined in the horizontal part of SPRxPOS.
- As long as the comparator remains enabled, the current contents of the sprite data register will be output at the selected horizontal position on a video line.
- The data in the sprite data registers does not change. It is either rewritten by the user or modified under DMA control.

The components described above produce the automatic DMA display as follows: When the sprites are in DMA mode, the 18-bit sprite pointer register (composed of SPRxPTH and SPRxPTL) is used to read the first two words from the sprite data structure. These words contain the starting and stopping position of the sprite. Next, the pointers write these words into SPRxPOS and SPRxCTL. After this write, the value in the pointers points to the address of the first data word (low word of data for line 1 of the sprite.)

Writing into the SPRxCTL register disabled the sprite. Now the sprite DMA channel will wait until the vertical beam counter value is the same as the data in the VSTART (Y value) part of SPRxPOS. When these values match, the system enables the sprite data access.

The sprite DMA channel examines the contents of VSTOP (from SPRxCTL, which is the location of the line after the last line of the sprite) and VSTART (from SPRxPOS) to see how many lines of sprite data are to be fetched. Two words are fetched per line of sprite height, and these words are written into the sprite data registers. The first word is stored in SPRxDATA and the second word in SPRxDATB.

The fetch and store for each horizontal scan line occurs during a horizontal blanking interval, far to the left of the start of the screen display. This arms the sprite horizontal comparators and allows them to start the output of the sprite data to the screen when the horizontal beam count value matches the value stored in the HSTART (X value) part of SPRxPOS.

If the count of VSTOP - VSTART equals zero, no sprite output occurs. The next data word pair will be fetched, but it will not be stored into the sprite data registers. It will instead become the next pair of data words for SPRxPOS and SPRxCTL.

When a sprite is used only once within a single display field, the final pair of data words, which follow the sprite color descriptor words, is loaded automatically as the next contents of the SPRxPOS and SPRxCTL registers. To stop the sprite after that first data set, the pair of words should contain all zeros.

Thus, if you have formed a sprite pattern in memory, this same pattern will be produced as pixels automatically under DMA control one line at a time.

Summary of Sprite Registers

There are eight complete sets of registers used to describe the sprites. Each set consists of five registers. Only the registers for sprite 0 are described here. All of the others are the same, except for the name of the register, which includes the appropriate number.

POINTERS

Pointers are registers that are used by the system to point to the *current* data being used. During a screen display, the registers are incremented to point to the data being used as the screen display progresses. Therefore, pointer registers must be freshly written during the start of the vertical blanking period.

SPR0PTH and SPR0PTL

This pair of registers contains the 18-bit word address of Sprite 0 DMA data. These registers contain the high three bits and low fifteen bits of the address, respectively. Because these two register addresses are contiguous, 68000 programmers can write a long word into SPR0PTH, as usual.

Pointer register names for the other sprites are:

SPR1PTH	SPR1PTL
SPR2PTH	SPR2PTL
SPR3PTH	SPR3PTL
SPR4PTH	SPR4PTL
SPR5PTH	SPR5PTL
SPR6PTH	SPR6PTL
SPR7PTH	SPR7PTL

CONTROL REGISTERS

SPR0POS

This is the sprite 0 position register. The word written into this register controls the position on the screen at which the upper left-hand corner of the sprite is to be placed. The most significant bit of the first data word will be placed in this position on the screen. Note that the sprites have a placement resolution on a full screen of 320 by 200. The sprite resolution is independent of the bit-plane resolution.

Bit positions:

- Bits 15-8 specify the vertical start position, bits V7 - V0.
- Bits 7-0 specify the horizontal start position, bits H8 - H1.

NOTE

> This register is normally only written by the sprite DMA channel itself. See the details above regarding the organization of the sprite data. This register is usually updated directly by DMA.

SPR0CTL

This register is normally used only by the sprite DMA channel. It contains control information that is used to control the sprite data-fetch process.

Bit positions:

- Bits 15-8 specify vertical stop position for a sprite image, bits V7 - V0.
- Bit 7 is the attach bit. This bit is valid only for odd-numbered sprites. It indicates that sprites 0, 1 (or 2,3 or 4,5 or 6,7) will, for color interpretation, be considered as paired, and as such will be called four bits deep. The odd-numbered (higher number) sprite contains bits with the higher binary significance.

  During attach mode, the attached sprites are normally moved horizontally and vertically together under processor control. This allows a greater selection of colors within the boundaries of the sprite itself. The sprites, although attached, remain capable of independent motion, however, and they will assume this larger color set only when their edges overlay one another.
- Bits 6-3 are reserved for future use (make zero).
- Bit 2 is bit V8 of vertical start.
- Bit 1 is bit V8 of vertical stop.
- Bit 0 is bit H0 of horizontal start.

Position and control registers for the other sprites are:

SPR1POS	SPR1CTL
SPR2POS	SPR2CTL
SPR3POS	SPR3CTL
SPR4POS	SPR4CTL
SPR5POS	SPR5CTL
SPR6POS	SPR6CTL
SPR7POS	SPR7CTL

DATA REGISTERS

The following registers, although defined in the address space of the main processor, are normally used only by the display processor. They are the holding registers for the data obtained by DMA cycles.

SPR0DATA, SPR0DATB	data registers for Sprite 0
SPR1DATA, SPR1DATB	data registers for Sprite 1
SPR2DATA, SPR2DATB	data registers for Sprite 2
SPR3DATA, SPR3DATB	data registers for Sprite 3
SPR4DATA, SPR4DATB	data registers for Sprite 4
SPR5DATA, SPR5DATB	data registers for Sprite 5
SPR6DATA, SPR6DATB	data registers for Sprite 6
SPR7DATA, SPR7DATB	data registers for Sprite 7

Summary of Sprite Color Registers

Sprite data words are used to select the color of the sprite pixels from the system color register set as indicated in the following tables.

If the bit combinations from single sprites are as shown in table 4-6, then the colors will be taken from the registers shown.

Table 4-6: Color Registers for Single Sprites

Single Sprites Sprite	 Value	Color Register
0 or 1	00	Not used *
	01	17
	10	18
	11	19
2 or 3	00	Not used *
	01	21
	10	22
	11	23
4 or 5	00	Not used *
	01	25
	10	26
	11	27
6 or 7	00	Not used *
	01	29
	10	30
	11	31

* Selects transparent mode.

If the bit combinations from attached sprites are as shown in table 4-7, then the colors will be taken from the registers shown.

Table 4-7: Color Registers for Attached Sprites

Attached Sprites

Value	**Color Register**
0000	Not used *
0001	17
0010	18
0011	19
0100	20
0101	21
0110	22
0111	23
1000	24
1001	25
1010	26
1011	27
1100	28
1101	29
1110	30
1111	31

* Selects transparent mode.

Chapter 5

AUDIO HARDWARE

Introduction

This chapter shows you how to directly access the audio hardware to produce sounds. The major topics in this chapter are:

- A brief overview of how a computer produces sound.
- How to produce simple steady and changing sounds and more complex ones.

- **How to use the audio channels for special effects, wiring them for stereo sound if desired, or using one channel to modulate another.**

- **How to produce quality sound within the system limitations.**

A section at the end of the chapter gives you values to use for creating musical notes on the equal-tempered musical scale.

This chapter is not a tutorial on computer sound synthesis; a thorough description of creating sound on a computer would require a far longer document. The purpose here is to point the way and show you how to use the Amiga's features. Computer sound production is fun but complex, and it usually requires a great deal of trial and error on the part of the user—you use the instructions to create some sound and play it back, readjust the parameters and play it again, and so on.

The following works are recommended for more information on creating music with computers:

- Wayne A. Bateman, *Introduction to Computer Music* (New York: John Wiley and Sons, 1980).

- Hal Chamberlain, *Musical Applications of Microprocessors* (Rochelle Park, New Jersey: Hayden, 1980).

INTRODUCING SOUND GENERATION

Sound travels through air to your ear drums as a repeated cycle of air pressure variations, or sound waves. Sounds can be represented as graphs that model how the air pressure varies over time. The attributes of a sound, as you hear it, are related to the shape of the graph. If the waveform is regular and repetitive, it will sound like a tone with steady pitch (highness or lowness), such as a single musical note. Each repetition of a waveform is called a cycle of the sound. If the waveform is irregular, the sound will have little or no pitch, like a loud clash or rushing water. How often the waveform repeats (its frequency) has an effect upon its pitch; sounds with higher frequencies are higher in pitch. Humans can hear sounds that have a frequency of between 20 and 20,000 cycles per second. The amplitude of the waveform (highest point on the graph), is related to the perceived loudness of the sound. Finally, the general shape of the waveform determines its tone quality, or timbre. Figure 5-1 shows a particular kind of waveform, called a sine wave, that represents one cycle of a simple tone.

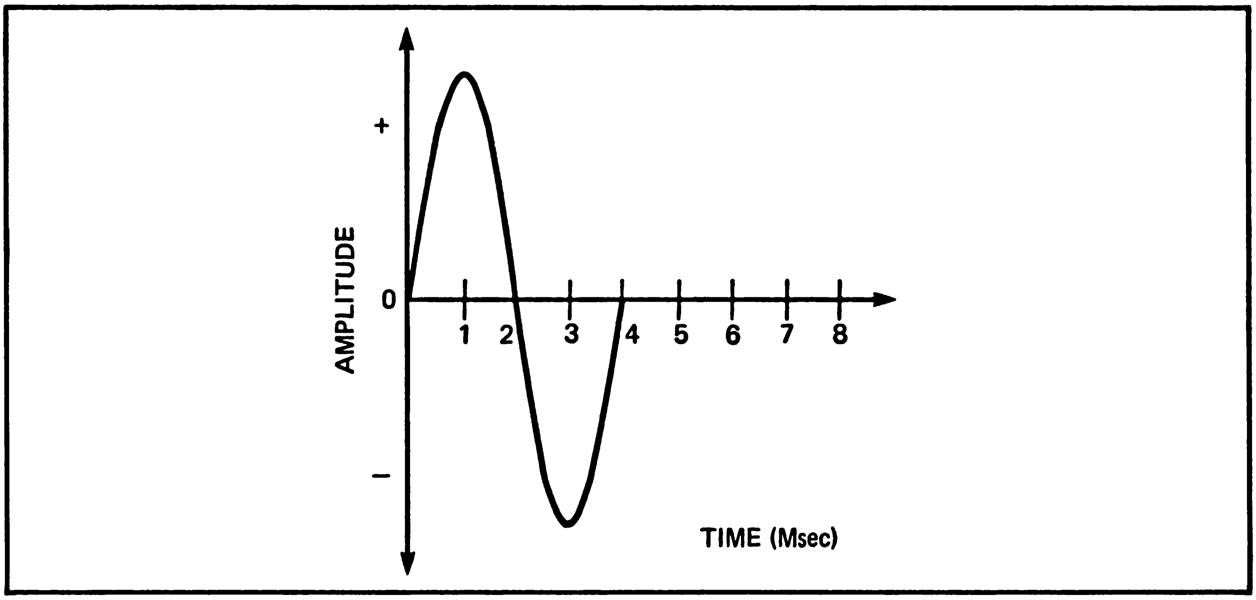


Figure 5-1: Sine Waveform

In electronic sound recording and output devices, the attributes of sounds are represented by the parameters of amplitude and frequency. Frequency is the number of cycles per second, and the most common unit of frequency is the Hertz (Hz), which is 1 cycle per second. Large values, or high frequencies, are measured in kilohertz (KHz) or megahertz (MHz).

Frequency is strongly related to the perceived pitch of a sound. When frequency increases, pitch rises. This relationship is exponential. An increase from 100 Hz to 200 Hz results in a large rise in pitch, but an increase from 1,000 Hz to 1,100 Hz is hardly noticeable. Musical pitch is represented in octaves. A tone that is one octave higher than another has a frequency twice as high as that of the first tone, and its perceived pitch is twice as high.

The second parameter that defines a waveform is its amplitude. In an electronic circuit, amplitude relates to the voltage or current in the circuit. When a signal is going to a speaker, the amplitude is expressed in watts. Perceived sound intensity is measured in decibels (db). Human hearing has a range of about 120 db; 1 db is the faintest audible sound. Roughly every 10 db corresponds to a doubling of sound, and 1 db is the smallest change in amplitude that is noticeable in a moderately loud sound. Volume, which is the amplitude of the sound signal which is output, corresponds logarithmically to decibel level.

The frequency and amplitude parameters of a sine wave are completely independent. When sound is heard, however, there is interaction between loudness and pitch. Lower-frequency sounds decrease in loudness much faster than high-frequency sounds.

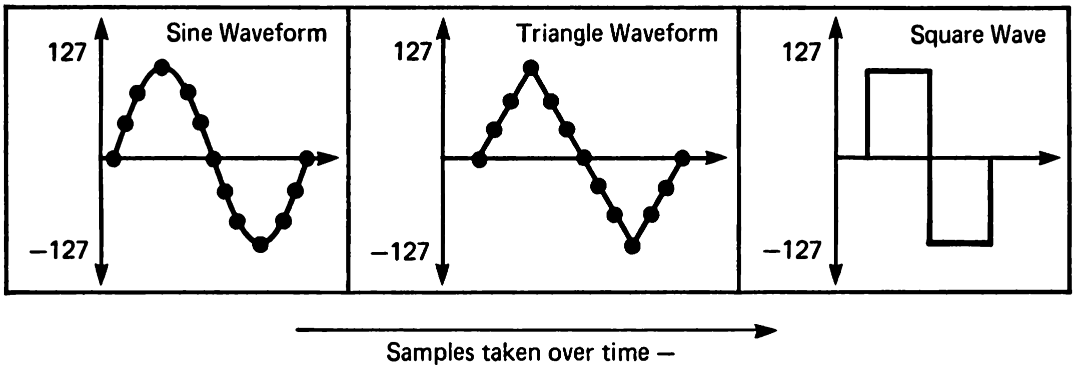
The third attribute of a sound, timbre, depends on the presence or absence of overtones, or harmonics. Any complex waveform is actually a mixture of sine waves of different amplitudes, frequencies, and phases (the starting point of the waveform on the time axis). These component sine waves are called harmonics. A square waveform, for example, has an infinite number of harmonics.

In summary, all steady sounds can be described by their frequency, overall amplitude, and relative harmonic amplitudes. The audible equivalents of these parameters are pitch, loudness, and timbre, respectively. Changing sound is a steady sound whose parameters change over time.

In electronic production of sound, an analog device, such as a tape recorder, records sound waveforms and their cycle frequencies as a continuously variable representation of air pressure. The tape recorder then plays back the sound by sending the waveforms to an amplifier where they are changed into analog voltage waveforms. The amplifier sends the voltage waveforms to a loudspeaker, which translates them into air pressure vibrations that the listener perceives as sound.

A computer cannot store analog waveform information. In computer production of sound, a waveform has to be represented as a finite string of numbers. This transformation is made by dividing the time axis of the graph of a single waveform into equal segments, each of which represents a short enough time so the waveform does not change a great deal. Each of the resulting points is called a sample. These samples are stored in memory, and you can play them back at a frequency that you determine. The computer feeds the samples to a digital-to-analog converter (DAC), which changes them into an analog voltage waveform. To produce the sound, the analog waveforms are sent first to an amplifier, then to a loudspeaker.

Figure 5-2 shows an example of a sine wave, a square wave, and a triangle wave, along with a table of samples for each. Note that the illustrations are not to scale and that there are fewer dots in the wave forms than there are samples in the table. The amplitude axis values 127 and -128 represent the high and low limits on relative amplitude.


127
Sine Waveform
−127
127
Triangle Waveform
−127
127
Square Wave
−127
Samples taken over time —

TIME	SINE	SQUARE	TRIANGLE
0	0	100	0
1	39	100	20
2	75	100	40
3	103	100	60
4	121	100	80
5	127	100	100
6	121	100	80
7	103	100	60
8	75	100	40
9	39	100	20
10	0	−100	0
11	−39	−100	−20
12	−75	−100	−40
13	−103	−100	−60
14	−121	−100	−80
15	−127	−100	−100
16	−121	−100	−80
17	−103	−100	−60
18	−75	−100	−40
19	−39	−100	−20

Figure 5-2: Digitized Amplitude Values

THE AMIGA SOUND HARDWARE

The Amiga has four hardware sound channels. You can independently program each of the channels to produce complex sound effects. You can also attach channels so that one channel modulates the sound of another or combine two channels for stereo effects.

Each audio channel includes an eight-bit digital-to-analog converter driven by a direct memory access (DMA) channel. The audio DMA can retrieve two data samples during each horizontal video scan line. For simple, steady tones, the DMA can automatically play a waveform repeatedly; you can also program all kinds of complex sound effects.

There are two methods of basic sound production on the Amiga — automatic (DMA) sound generation and direct (non-DMA) sound generation. When you use automatic sound generation, the system retrieves data automatically by direct memory access.

Forming and Playing a Sound

This section shows you how to create a simple, steady sound and play it. Many basic concepts that apply to all sound generation on the Amiga are introduced in this section.

To produce a steady tone, follow these basic steps:

1. Decide which channel to use.
2. Define the waveform and create the sample table in memory.
3. Set registers telling the system where to find the data and the length of the data.
4. Select the volume at which the tone is to be played.
5. Select the sampling period, or output rate of the data.
6. Select an audio channel and start up the DMA.

DECIDING WHICH CHANNEL TO USE

The Amiga has four audio channels. Channels 0 and 3 are connected to the left-side stereo output jack. Channels 1 and 2 are connected to the right-side output jack. Select a channel on the side from which the output is to appear.

CREATING THE WAVEFORM DATA

The waveform used as an example in this section is a simple sine wave, which produces a pure tone. To conserve memory, you normally define only one full cycle of a waveform in memory. For a steady, unchanging sound, the values at the waveform's beginning and ending points and the trend or slope of the data at the beginning and end should be closely related. This ensures that a continuous repetition of the waveform sounds like a continuous stream of sound.

Sound data is organized as a set of eight-bit data items; each item is a sample from the waveform. Each data word retrieved for the audio channel consists of two samples. Sample values can range from -128 to +127.

As an example, the data set shown below produces a close approximation to a sine wave. Note that the data is stored in byte address order with the first digitized amplitude value at the lowest byte address, the second at the next byte address, and so on. Also, note that the first byte of data must start at a word-address boundary. This is because the audio DMA retrieves one word (16 bits) at a time and uses the sample it reads as two bytes of data.

To use audio channel 0, write the address of "audiodata" into AUD0LC, where the audio data is organized as shown below. For simplicity, "AUDxLC" in the table below stands for the combination of the two actual location registers (AUDxLCH and AUDxLCL). For the audio DMA channels to be able to retrieve the data, the data address to which AUD0LC points must be located in the low 512K bytes of RAM.

Table 5-1: Sample Audio Data Set for Channel 0

audiodata --->	AUD0LC *	100	98
	AUD0LC + 2 **	92	83
	AUD0LC + 4	71	56
	AUD0LC + 6	38	20
	AUD0LC + 8	0	-20
	AUD0LC + 10	-38	-56
	AUD0LC + 12	-71	-83
	AUD0LC + 14	-92	-83
	AUD0LC + 16	-100	-98
	AUD0LC + 18	-92	-83
	AUD0LC + 20	-71	-56
	AUD0LC + 22	-38	-20
	AUD0LC + 24	0	20
	AUD0LC + 26	38	56
	AUD0LC + 28	71	83
	AUD0LC + 30	92	98

Notes

* Audio data is located on a word-address boundary.

** AUD0LC stands for AUD0LCL and AUD0LCH.

TELLING THE SYSTEM ABOUT THE DATA

In order to retrieve the sound data for the audio channel, the system needs to know where the data is located and how long (in words) the data is.

The location registers AUDxLCH and AUDxLCL contain the high three bits and the low fifteen bits, respectively, of the starting address of the audio data. Since these two register addresses are contiguous, writing a long word into AUDxLCH moves the audio data address into both locations. The "x" in the register names stands for the number of the audio channel where the output will occur. The channels are numbered 0, 1, 2, and 3.

These registers are *location* registers, as distinguished from *pointer* registers. You need to specify the contents of these registers only once; no resetting is necessary when you wish the audio channel to keep on repeating the same waveform. Each time the system retrieves the last audio word from the data area, it uses the contents of these location registers to again find the start of the data. Assuming the first word of data starts at location "audiodata" and you are using channel 0, here is how to set the location registers:

```
        AUD0LC  EQU     AUD0LCH         ;AUD0LC stands for AUD0LCL

WHERE0DATA:

                LEA     AUDIODATA, A0
                MOVE.L  A0, AUD0LC      ;Put address (32 bits)
                                        ; into location register.
```

The length of the data is the number of samples in your waveform divided by 2, or the number of words in the data set. Using the sample data set above, the length of the data is 16 words. You write this length into the audio data length register for this channel. The length register is called AUDxLEN, where "x" refers to the channel number. You set the length register AUD0LEN to 16 as shown below.

```
        SETAUD0LENGTH:  MOVE.W #16, AUD0LEN
```

SELECTING THE VOLUME

The volume you set here is the overall volume of all the sound coming from the audio channel. The relative loudness of sounds, which will concern you when you combine notes, is determined by the amplitude of the wave form. There is a six-bit volume register for each audio channel. To control the volume of sound that will be output through the selected audio channel, you write the desired value into the register AUDxVOL, where "x" is replaced by the channel number. You can specify values from 64 to 0. These volume values correspond to decibel levels. At the end of this chapter is a table showing the decibel value for each of the 65 volume levels. For a typical output at volume 64, with maximum data values of -128 to 127, the voltage output is between +.4 volts and -.4 volts. Some volume levels and the corresponding decibel values are shown in table 5-2.

Table 5-2: Volume Values

Volume	Decibel Value	
64	0	(maximum volume)
48	-2.5	
32	-6.0	
16	-12.0	(12 db down from the volume at maximum level)

For any volume setting from 64 to 0, you write the value into bits 5-0 of AUD0VOL. For example:

```
SET AUD0VOLUME:    MOVE.W #48, AUD0VOL
```

The decibels are shown as negative values from a maximum of 0 because this is the way a recording device, such as a tape recorder, shows the recording level. Usually, the recorder has a dial showing 0 as the optimum recording level. Anything less than the optimum value is shown as a minus quantity.

SELECTING THE DATA OUTPUT RATE

The pitch of the sound produced by the waveform depends upon its frequency. To tell the system what frequency to use, you need to specify the sampling period. The sampling period specifies the number of system clock ticks, or timing intervals, that should elapse between each sample (byte of audio data) fed to the digital-to-analog converter in the audio channel. There is a period register for each audio channel. The value of the period register is used for count-down purposes; each time the register counts down to 0, another sample is retrieved from the waveform data set for output. In units, the period value represents clock ticks per sample. The minimum period value you should use is 124 ticks per sample and the maximum is 65535. For high-quality sound, there are other constraints on the sampling period (see the section called "Producing High-quality Sound"). Note that a low period value corresponds to a higher frequency sound and a high period value corresponds to a lower frequency sound.

Limitations on Selection of Sampling Period

The sampling period is limited by the number of DMA cycles allocated to an audio channel. Each audio channel is allocated one DMA slot per horizontal scan line of the screen display. An audio channel can retrieve two data samples during each horizontal scan line. The following calculation gives the maximum sampling rate in samples per second.

2 samples/line X 262.5 lines/frame X 59.94 frames/second = 31,469 samples/second

The figure of 31,469 is a theoretical maximum. In order to save buffers, the hardware is designed to handle 28,867 samples/second. The system timing interval is 279.365 nanoseconds, or .279365 microseconds. The maximum sampling rate of 28,867 samples per second is 34.642 microseconds per sample (1/28,867 = .000034642). The formula for calculating the sampling period is

Period value = sample interval/clock interval

Thus, the minimum period value is derived by dividing 34.642 microseconds per sample by the number of microseconds per interval:

$$\begin{aligned}\text{Minimum period} &= \frac{\text{34.642 microseconds/sample}}{\text{0.279365 microseconds/interval}} \\ &= \text{124 timing intervals/sample}\end{aligned}$$

Therefore, a value of at least 124 must be written into the period register to assure that the audio system DMA will be able to retrieve the next data sample. If the period value is below 124, by the time the cycle count has reached 0, the audio DMA will not have had enough time to retrieve the next data sample and the previous sample will be reused.

Specifying the Period Value

After you have selected the desired interval between data samples, you can calculate the value to place in the period register by using the period formula:

Period value = desired interval/clock interval

As an example, say you wanted to produce a 1 KHz sine wave, using a table of eight data samples (four data words) (see figure 5-3).

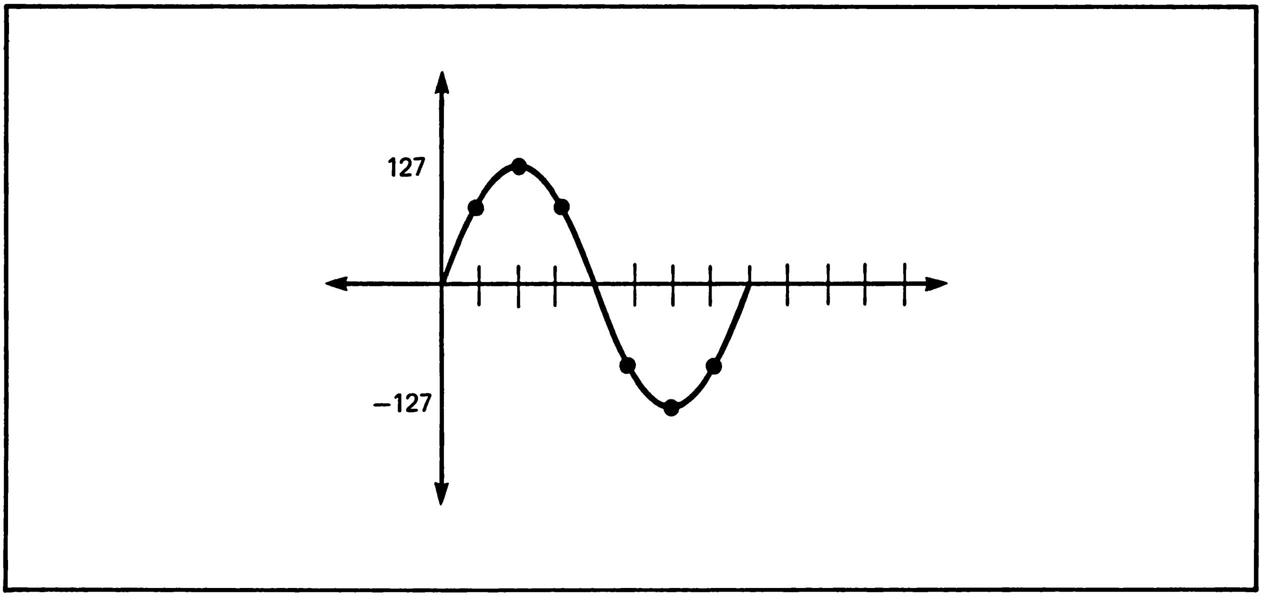


Figure 5-3: Example Sine Wave

Sampled Values: 0
90
127
90
0
-90
-127
-90

To output the series of eight samples at 1 KHz (1,000 cycles per second), each full cycle is output in 1/1000th of a second. Therefore, each individual value must be retrieved in 1/8th of that time. This translates to 1,000 microseconds per waveform or 125 microseconds per sample. To correctly produce this waveform, the period value should be

$$\text{Period value} = \frac{\text{125 microseconds/sample}}{\text{0.279365 microseconds/interval}}$$

$$= \text{447 timing intervals/sample}$$

To set the period register, you must write the period value into the register AUDxPER, where "x" is the number of the channel you are using. For example, the following instruction shows how to write a period value of 447 into the period register for channel 0.

```
SETAUD0PERIOD:   MOVE.W   #447, AUD0PER
```

To produce high-quality sound, avoiding aliasing distortion, you should observe the limitations on period values that are discussed in the section below called "Producing Quality Sound."

For the relationship between period and musical pitch, see the section at the end of the chapter, which contains a listing of the equal-tempered musical scale.

PLAYING THE WAVEFORM

After you have defined the audio data location, length, volume and period, you can play the waveform by starting the DMA for that audio channel. This starts the output of sound. Once started, the DMA continues until you specifically stop it. Thus, the waveform is played over and over again, producing the steady tone. The system uses the value in the location registers each time it replays the waveform.

To start the channel, you write a 1 into the AUDxEN bit of the DMA control register named DMACON. To start the DMA, you write a 1 into the DMAEN bit of DMACON. All these bits and their meanings are shown in table 5-3.

Table 5-3: DMA and Audio Channel Enable Bits

DMACON Register

Bit	Name	Function
15	SETCLR	When this bit is written as a 1, it sets any bit in DMACONW for which the corresponding bit position is also a 1, leaving all other bits alone.
9	DMAEN	Only while this bit is a 1 can *any* direct memory access occur.
3	AUD3EN	Audio channel 3 enable.
2	AUD2EN	Audio channel 2 enable.
1	AUD1EN	Audio channel 1 enable.
0	AUD0EN	Audio channel 0 enable.

For example, if you are using channel 0, then you write a 1 into bit 9 to enable DMA and a 1 into bit 0 to enable the audio channel, as shown below.

```
SET       EQU     $08000
AUD0EN    EQU     $01
DMAEN     EQU     $0200

BEGINCHAN0:

          MOVE.W   #(SET + AUD0EN + DMAEN), DMACONW
```

STOPPING THE AUDIO DMA

You can stop the channel by writing a 0 into the AUDxEN bit at any time. However, you cannot resume the output at the same point in the waveform by just writing a 1 in the bit again. Enabling an audio channel almost always starts the data output again from the top of the list of data pointed to by the location registers for that channel. If the channel is disabled for a very short time (less than two sampling periods) it may stay on and thus continue from where it left off.

The following example shows how to stop audio DMA for one channel.

```
        CLEAR    EQU     0

STOPAUDCHAN0:
                         MOVE.W   #(CLEAR + AUD0EN), DMACONW
```

SUMMARY

These are the steps necessary to produce a steady tone:

1. Define the waveform.
2. Create the data set containing the pairs of data samples (data words). Normally, a data set contains the definition of one waveform.
3. Set the location registers:

   AUDxLCH (high three bits)

   AUDxLCL (low fifteen bits)

4. Set the length register, AUDxLEN, to the number of data words to be retrieved before starting at the address currently in AUDxLC.
5. Set the volume register, AUDxVOL.
6. Set the period register, AUDxPER
7. Start the audio DMA by writing a 1 into bit 9, DMAEN, along with a 1 in the SETCLR bit and a 1 in the position of the AUDxEN bit of the channel or channels you want to start.

EXAMPLE

In this example, which gathers together all of the program segments from the preceding sections, a sine wave is played through channel 0.

```
        AUD0LC  EQU      AUD0LCH
        SET     EQU      $08000
        CLEAR   EQU      0
        AUD0EN  EQU      $01
        DMAEN   EQU      $0200

                DS.W     0                  ;Be sure word-aligned
SINEDATA:
                DC.B     0, 90, 127, 90, 0, -90, -127, -90
MAIN:
                LEA      SINEDATA, A0       ;Address of data to
                                            ; audio location register 0
WHERE0DATA:
                MOVE.L   A0, AUD0LC         ;The 68000 writes
                                            ; this as though it were
                                            ; a 32-bit register at the
                                            ; low-bits location
                                            ; (common to all locations
                                            ; and pointer registers
                                            ; in the system).


SETAUD0LENGTH:
                MOVE.W   #4, AUD0LEN        ;Set length in words

SETAUD0VOLUME:
                MOVE.W   #64, AUD0VOL       ;Use maximum volume

SETAUD0PERIOD:
                MOVE.W   #447, AUD0PER

BEGINCHAN0:
                MOVE.W   #(SET + DMAEN + AUD0EN), DMACONW

                END
```

Producing Complex Sounds

In addition to simple tones, you can create more complex sounds, such as different musical notes joined into a one-voice melody, different notes played at the same time, or modulated sounds.

JOINING TONES

Tones are joined by writing the location and length registers, starting the audio output, and rewriting the registers in preparation for the next audio waveform that you wish to connect to the first one. This is made easy by the timing of the audio interrupts and the existence of back-up registers. The location and length registers are read by the DMA channel before audio output begins. The DMA channel then stores the values in back-up registers. Once the original registers have been read by the DMA channel, you can change their values without disturbing the operation you started with the original register contents. Thus, you can write the contents of these registers, start an audio output, and then rewrite the registers in preparation for the next waveform you want to connect to this one.

Interrupts occur immediately after the audio DMA channel has read the location and length registers and stored their values in the back-up registers. Once the interrupt has occurred, you can rewrite the registers with the location and length for the next waveform segment. This combination of back-up registers and interrupt timing lets you keep one step ahead of the audio DMA channel, allowing your sound output to be continuous and smooth.

If you do not rewrite the registers, the current waveform will be repeated. Each time the length counter reaches zero, both the location and length registers are reloaded with the same values to continue the audio output.

Example

This example details the system audio DMA action in a step-by-step fashion.

Suppose you wanted to join together a sine and a triangle waveform, end-to-end, for a special audio effect, alternating between them. The following sequence shows the action of your program as well as its interaction with the audio DMA system. The example assumes that the period, volume, and length of the data set remains the same for the sine wave and the triangle wave.

```
If (wave = triangle)
        write AUD0LCL with address of sine wave data.

Else if (wave = sine)
        write AUD0LCL with address of triangle wave data.
```

Main Program

1. Set up volume, period, and length.

2. Write AUD0LCL with address of sine wave data.

3. Start DMA.

4. Continue with something else.

System Response

As soon as DMA starts,

a. Copy to "back-up" length register from AUD0LEN.

b. Copy to "back-up" location register from AUD0LCL (will be used as a pointer showing current data word to fetch).

c. Create an interrupt for the 68000 saying that it has completed retrieving working copies of length and location registers.

d. Start retrieving audio data each allocated DMA time slot.

PLAYING MULTIPLE TONES AT THE SAME TIME

You can play multiple tones either by using several channels independently or by summing the samples in several data sets, playing the summed data sets through a single channel.

Since all four audio channels are independently programmable, each channel has its own data set; thus a different tone or musical note can be played on each channel.

MODULATING SOUND

To provide more complex audio effects, you can use one audio channel to modulate another. This increases the range and type of effects that can be produced. You can modulate a channel's frequency or amplitude, or do both types of modulation on a channel at the same time.

Amplitude modulation affects the volume of the waveform. It is often used to produce vibrato or tremolo effects. Frequency modulation affects the period of the waveform. Although the basic waveform itself remains the same, the pitch is increased or decreased by frequency modulation.

The system uses one channel to modulate another when you attach two channels. The attach bits in the ADKCON register control how the data from an audio channel is interpreted (see the table below). Normally, each channel produces sound when it is enabled. If the "attach" bit for an audio channel is set, that channel ceases to produce sound and its data is used to modulate the sound of the next higher-numbered channel. When a channel is used as a modulator, the words in its data set are no longer treated as two individual bytes. Instead, they are used as "modulator" words. The data words from the *modulator* channel are written into the corresponding registers of the *modulated* channel each time the period register of the modulator channel times out.

To modulate only the amplitude of the audio output, you must attach a channel as a volume modulator. Define the modulator channel's data set as a series of words, each containing volume information in the following format:

Bits	**Function**
15 - 7	Not used
6 - 0	Volume information, V6 - V0

To modulate only the frequency, you must attach a channel as a period modulator. Define the modulator channel's data set as a series of words, each containing period information in the following format:

Bits	**Function**
15 - 0	Period information, P15 - P0

If you want to modulate both period and volume on the same channel, you need to attach the channel as both a period and volume modulator. For instance, if channel 0 is used to modulate both the period and frequency of channel 1, you set two attach bits — bit 0 to modulate the volume and bit 4 to modulate the period. When period and volume are both modulated, words in the modulator channel's data set are defined alternately as volume and period information.

The sample set of data in table 5-4 shows the differences in interpretation of data when a channel is used directly for audio, when it is attached as volume modulator, when it is attached as a period modulator, and when it is attached as a modulator of both volume and period.

Table 5-4: Data Interpretation in Attach Mode

<table>
<tr><th>Data Words</th><th>Independent (not Modulating)</th><th>Modulating Both Period and Volume</th><th>Modulating Period Only</th><th>Modulating Volume Only</th></tr>
<tr><td>Word 1</td><td>| data | data |</td><td>| volume for other channel |</td><td>| period |</td><td>| volume |</td></tr>
<tr><td>Word 2</td><td>| data | data |</td><td>| period for other channel |</td><td>| period |</td><td>| volume |</td></tr>
<tr><td>Word 3</td><td>| data | data |</td><td>| volume for other channel |</td><td>| period |</td><td>| volume |</td></tr>
<tr><td>Word 4</td><td>| data | data |</td><td>| period for other channel |</td><td>| period |</td><td>| volume |</td></tr>
</table>

The lengths of the data sets of the modulator and the modulated channels are completely independent.

Channels are attached by the system in a predetermined order, as shown in table 5-5. To attach a channel as a modulator, you set its attach bit to 1. If you set either the volume or period attach bits for a channel, that channel's audio output will be disabled; the channel will be attached to the next higher channel, as shown in table 5-5. Because an attached channel always modulates the next higher numbered channel, you cannot attach channel 3. Writing a 1 into channel 3's modulate bits only disables its audio output.

Table 5-5: Channel Attachment for Modulation

ADKCON Register

Bit	Name	Function
7	ATPER3	Use audio channel 3 to modulate nothing (disables audio output of channel 3)
6	ATPER2	Use audio channel 2 to modulate *period* of channel 3
5	ATPER1	Use audio channel 1 to modulate *period* of channel 2
4	ATPER0	Use audio channel 0 to modulate *period* of channel 1
3	ATVOL3	Use audio channel 3 to modulate nothing (disables audio output of channel 3)
2	ATVOL2	Use audio channel 2 to modulate *volume* of channel 3
1	ATVOL1	Use audio channel 1 to modulate *volume* of channel 2
0	ATVOL0	Use audio channel 0 to modulate *volume* of channel 1

Producing High-quality Sound

When trying to create high-quality sound, you need to consider the following factors:

- Waveform transitions.
- Sampling rate.
- Efficiency.
- Noise reduction.
- Avoidance of aliasing distortion.
- Limitations of the low pass filter.

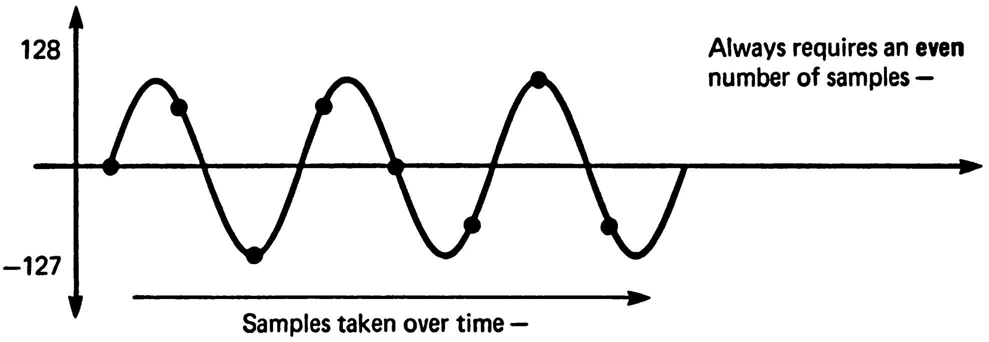
MAKING WAVEFORM TRANSITIONS

To avoid unpleasant sounds when you change from one waveform to another, you need to make the transitions smooth. You can avoid "clicks" by making sure the waveforms start and end at approximately the same value. You can avoid "pops" by starting a waveform only at a zero-crossing point. You can avoid "thumps" by arranging the average amplitude of each wave to be about the same value. The average amplitude is the sum of the bytes in the waveform divided by the number of bytes in the waveform.

SAMPLING RATE

If you need high precision in your frequency output, you may find that the frequency you wish to produce is somewhere between two available sampling rates, but not close enough to either rate for your requirements. In those cases, you may have to adjust the length of the audio data table in addition to altering the sampling rate.

For higher frequencies, you may also need to use audio data tables that contain more than one full cycle of the audio waveform to reproduce the desired frequency more accurately, as illustrated in figure 5-4.


128
–127
Always requires an **even** number of samples –
Samples taken over time –

Shows a case in which a high-frequency waveform may need more than one full cycle to accurately reproduce the periodic waveform

Figure 5-4: Waveform with Multiple Cycles

EFFICIENCY

A certain amount of overhead is involved in the handling of audio DMA. If you are trying to produce a smooth continuous audio synthesis, you should try to avoid as much of the system control overhead as possible. Basically, the larger the audio buffer you provide to the system, the less often it will need to interrupt to reset the pointers to the top of the next buffer and, coincidentally, the lower the amount of system interaction that will be required. If there is only one waveform buffer, the hardware automatically resets the pointers, so no software overhead is used for resetting them.

The "Joining Tones" section illustrated how you could join "ends" of tones together by responding to interrupts and changing the values of the location registers to splice tones together. If your system is heavily loaded, it is possible that the response to the interrupt might not happen in time to assure a smooth audio transition. Therefore, it is advisable to utilize the longest possible audio table where a smooth output is required. This takes advantage of the audio DMA capability as well as minimizing the number of interrupts to which the 68000 must respond.

NOISE REDUCTION

To reduce noise levels and produce an accurate sound, try to use the full range of -128 to 127 when you represent a waveform. This reduces how much noise (quantization error) will be added to the signal by using more bits of precision. Quantization noise is caused by the introduction of round-off error. If you are trying to reproduce a signal, such as a sine wave, you can represent the amplitude of each sample with only so many digits of accuracy. The difference between the real number and your approximation is round-off error, or noise.

By doubling the amplitude, you create half as much noise because the size of the steps of the wave form stays the same and is therefore a smaller fraction of the amplitude. In other words, if you try to represent a waveform using, for example, a range of only +3 to -3, the size of the error in the output would be considerably larger than if you use a range of +127 to -128 to represent the same signal. Proportionally, the digital value used to represent the waveform amplitude will have a lower error. As you increase the number of possible sample levels, you decrease the relative size of each step and, therefore, decrease the size of the error.

To produce quiet sounds, continue to define the waveform using the full range, but adjust the volume. This maintains the same level of accuracy (signal-to-noise ratio) for quiet sounds as for loud sounds.

ALIASING DISTORTION

When you use sampling to produce a waveform, a side effect is caused when the sampling rate "beats" or combines with the frequency you wish to produce. This produces two additional frequencies, one at the sampling rate plus the desired frequency and the other at the sampling rate minus the desired frequency. This phenomenon is called aliasing distortion.

Aliasing distortion is eliminated when the sampling rate exceeds the output frequency by at least 7 KHz. This puts the beat frequency outside the range of the low-pass filter, cutting off the undesirable frequencies. Figure 5-5 shows a frequency domain plot of the anti-aliasing low-pass filter used in the system.

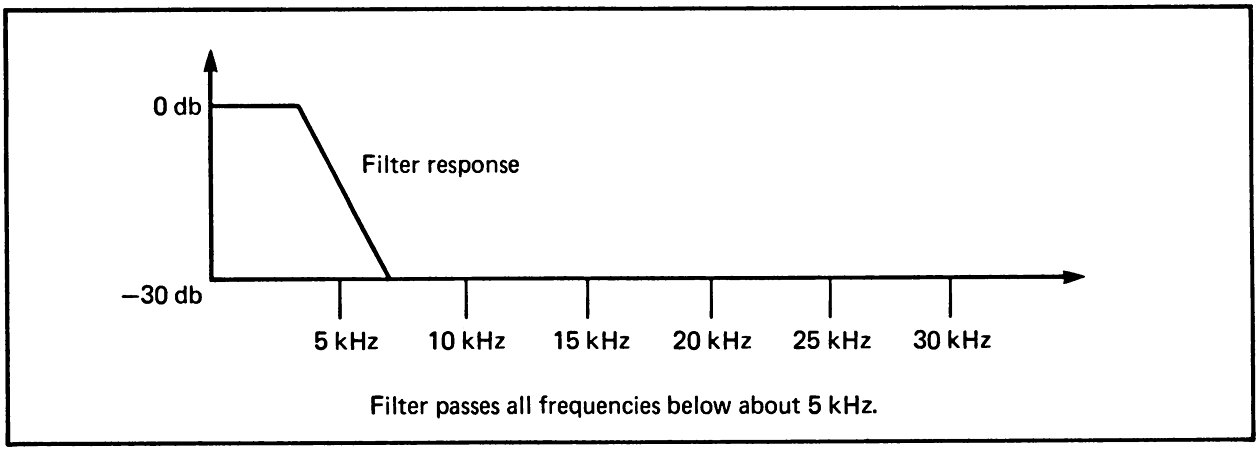


Figure 5-5: Frequency Domain Plot of Low-Pass Filter

Figure 5-6 shows that it is permissible to use a 12 KHz sampling rate to produce a 4 KHz waveform. Both of the beat frequencies are outside the range of the filter, as shown in these calculations:

12 + 4 = 16 KHz
12 - 4 = 8 KHz

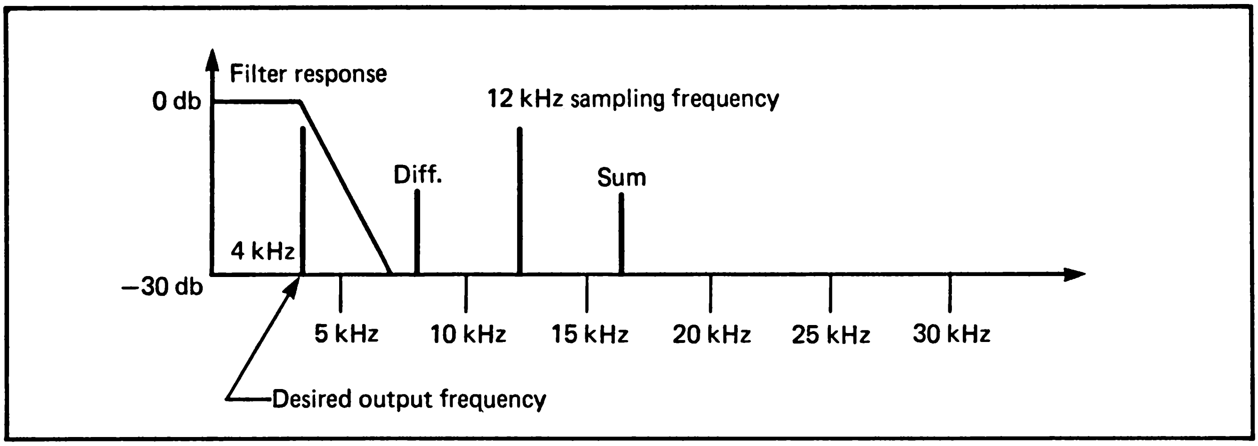


Figure 5-6: Noise-free Output (No Aliasing Distortion)

You can see in figure 5-7 that is unacceptable to use a 10 KHz sampling rate to produce a 4 KHz waveform. One of the beat frequencies (10 - 4) is within the range of the filter, allowing some of that undesirable frequency to show up in the audio output.

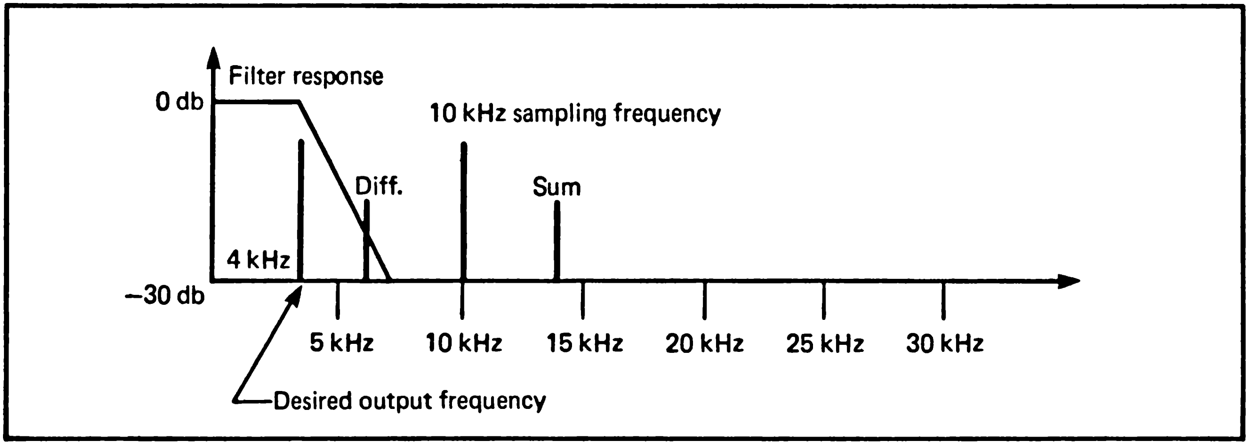


Figure 5-7: Some Aliasing Distortion

All of this gives rise to the following equation, showing that the sampling frequency must exceed the output frequency by at least 7 KHz, so that the beat frequency will be above the cutoff range of the anti-aliasing filter:

$$\text{Minimum sampling rate} = \text{highest frequency component} + 7\ \text{KHz}$$

The frequency component of the equation is stated as "highest frequency component" because you may be producing a complex waveform with multiple frequency elements, rather than a pure sine wave.

LOW-PASS FILTER

The system includes a low-pass filter that eliminates aliasing distortion as described above. This filter becomes active around 4 KHz and gradually begins to attenuate (cut off) the signal. Generally, you cannot clearly hear frequencies higher than 7 KHz. Therefore, you get the most complete frequency response in the frequency range of 0 - 7 KHz. If you are making frequencies from 0 to 7 KHz, you should select a sampling rate no less than 14 KHz, which corresponds to a sampling period in the range 124 to 256.

At a sampling period around 320, you begin to lose the higher frequency values between 0 KHz and 7 KHz, as shown in table 5-6.

Table 5-6: Sampling Rate and Frequency Relationship

	Sampling Period	**Sampling Rate (KHz)**	**Maximum Output Frequency (KHz)**
Maximum sampling rate	124	29	7
Minimum sampling rate for 7 KHz output	256	14	7
Sampling rate too low for 7 KHz output	320	11	4

Using Direct (Non-DMA) Audio Output

It is possible to create sound by writing audio data one word at a time to the audio output addresses, instead of setting up a list of audio data in memory. This method of controlling the output is more processor-intensive and is therefore not recommended.

To use direct audio output, do not enable the DMA for the audio channel you wish to use; this changes the timing of the interrupts. The normal interrupt occurs after a data address has been read; in direct audio output, the interrupt occurs after one data word has been output.

Unlike in the DMA-controlled automatic data output, in direct audio output, if you do not write a new set of data to the output addresses before two sampling intervals have elapsed, the audio output will cease changing. The last value remains as an output of the digital-to-analog converter.

The volume and period registers are set as usual.

The Equal-tempered Musical Scale

This section gives a close approximation of the equal-tempered scale. The "Period" column gives the period count you enter into the period register.

See the explanatory notes following this table for determining AUDxLEN value.

Table 5-7: The Equal-tempered Scale

Period	**Note**	**Ideal Frequency** (with AUDxLEN=8)	**Actual Frequency** (with AUDxLEN=8)
508	A	440.0	440.4
480	A#	466.2	466.1
453	B	493.9	493.9
428	C	523.3	522.7
404	C#	554.4	553.8
381	D	587.3	587.2
360	D#	622.3	621.4
339	E	659.3	659.9
320	F	698.5	699.1
302	F#	740.0	740.8
285	G	784.0	785.0
269	G#	830.6	831.7
254	A	880.0	880.8
240	A#	932.3	932.2
226	B	987.8	989.9
214	C	1046.5	1045.4
202	C#	1108.7	1107.5
190	D	1174.7	1177.5
180	D#	1244.5	1242.9
170	E	1318.5	1316.0
160	F	1396.9	1398.3
151	F#	1480.0	1481.6
143	G	1568.0	1564.5
135	G#	1661.2	1657.2

Notes for table 5-7:

In this scale, the frequency for the note A is 440.0 Hz and A# is the twelfth root of 2 (1.059463) times higher in frequency than A. The note B is the twelfth root of 2 higher than A#. This is followed by C, C#, D, D#, E, F, F#, G, and G#, and goes back to A at 880.0 Hz, an octave higher, and so on. Use this scale for waveforms where the fundamental is 2 to the *n*th bytes long and where *n* is an integer. For example, for A at 440.0 Hz with a period of 508, the sample table contains 16 samples per cycle:

$$\frac{3579545 \text{ clocks/second}}{508 \text{ clocks/sample X } 440 \text{ cycles/second}} = 16 \text{ samples/cycle}$$

$$= 2^4$$

$$n = 4$$

It follows that for A at 440.0 Hz with a period of 254, the sample table has to contain 32 samples per cycle (AUDxLEN = 16).

The general rule is that doubling the sampling frequency (halving the sampling period) changes the octave of the note being played. Thus, if you play a C at a sampling period of 256, then playing the same note with a sampling period of 128 gives a C an octave higher.

Before using the lower octaves in this table, be sure to read the section called "Aliasing Distortion."

Decibel Values for Volume Ranges

Table 5-8 provides the corresponding decibel values for the volume ranges of the Amiga system.

Table 5-8: Decibel Values and Volume Ranges

Volume	Decibel Value
64	0.0
63	-0.1
62	-0.3
61	-0.4
60	-0.6
59	-0.7
58	-0.9
57	-1.0
56	-1.2
55	-1.3
54	-1.5
53	-1.6
52	-1.8
51	-2.0
50	-2.1
49	-2.3
48	-2.5
47	-2.7
46	-2.9
45	-3.1
44	-3.3
43	-3.5
42	-3.7
41	-3.9
40	-4.1
39	-4.3
38	-4.5
37	-4.8
36	-5.0
35	-5.2
34	-5.5
33	-5.8
32	-6.0
31	-6.3
30	-6.6
29	-6.9
28	-7.2
27	-7.5
26	-7.8
25	-8.2
24	-8.5
23	-8.9
22	-9.3
21	-9.7
20	-10.1
19	-10.5
18	-11.0
17	-11.5
16	-12.0
15	-12.6
14	-13.2
13	-13.8
12	-14.5
11	-15.3
10	-16.1
9	-17.0
8	-18.1
7	-19.2
6	-20.6
5	-22.1
4	-24.1
3	-26.6
2	-30.1
1	-36.1
0	Minus infinity

The Audio State Machine

For an explanation of the various states, refer to figure 5-8. There is one audio state machine for each channel. The machine has eight states and is clocked at the system clock frequency of 3.58 MHz. Three of the states are basically unused and just transfer back to the idle (000) state. One of the paths out of the idle state is designed for interrupt-driven operation (processor provides the data), and the other path is designed for DMA-driven operation (the "Agnus" special chip provides the data).

In interrupt-driven operation, transfer to the main loop (states 010 and 011) occurs immediately after data is written by the processor. In the 010 state the upper byte is output, and in the 011 state the lower byte is output. Transitions such as 010→011→010 occur whenever the period counter counts down to one. The period counter is reloaded at these transitions. As long as the interrupt is cleared by the processor in time, the machine remains in the main loop. Otherwise, it enters the idle state. Interrupts are generated on every word transition (011→010).

In DMA-driven operation, transition to the 001 state occurs and DMA requests are sent to Agnus as soon as DMA is turned on. Because of pipelining in Agnus, the first data word must be thrown away. State 101 is entered as soon as this word arrives; a request for the next data word has already gone out. When the data arrives, state 010 is entered and the main loop continues until the DMA is turned off. The length counter counts down once with each word that comes in. When it finishes, a DMA restart request goes to Agnus along with the regular DMA request. This tells Agnus to reset the pointer to the beginning of the table of data. Also, the length counter is reloaded and an interrupt request goes out soon after the length counter finishes (counts to one). The request goes out just as the last word of the waveform starts its output.

DMA requests and restart requests are transferred to Agnus once each horizontal line, and the data comes back about 14 clock cycles later (the duration of a clock cycle is 280 ns).

In attach mode, things run a little differently. In attach volume, requests occur as they do in normal operation (on the 011→010) transition). In attach period, a set of requests occurs on the 010→011 transition. When both attach period and attach volume are high, requests occur on both transitions.

If the sampling rate is set much higher than the normal maximum sampling rate (approximately 29 KHz), the two samples in the buffer register will be repeated. If the filter on the Amiga is bypassed and the volume is set to the maximum ($40), this feature can be used to make modulated carriers up to 1.79 MHz. The modulation is placed in the memory map, with plus values in the even bytes and minus values in the odd bytes.

The symbols used in the state diagram are explained in the following list. Upper-case names indicate external signals; lower-case names indicate local signals.

AUDxON	DMA on "x" indicates channel number (signal from DMACON).
AUDxIP	Audio interrupt pending (input to channel from interrupt circuitry).
AUDxIR	Audio interrupt request (output from channel to interrupt circuitry)
intreq1	Interrupt request that combines with intreq2 to form AUDxIR..
intreq2	Prepare for interrupt request. Request comes out after the next 011→010 transition in normal operation.
AUDxDAT	Audio data load signal. Loads 16 bits of data to audio channel.
AUDxDR	Audio DMA request to Agnus for one word of data.
AUDxDSR	Audio DMA request to Agnus to reset pointer to start of block.
dmasen	Restart request enable.
percntrld	Reload period counter from back-up latch typically written by processor with AUDxPER (can also be written by attach mode).
percount	Count period counter down one latch.
perfin	Period counter finished (value = 1).
lencntrld	Reload length counter from back-up latch.
lencount	Count length counter down one notch.
lenfin	Length counter finished (value = 1).
volcntrld	Reload volume counter from back-up latch.
pbufld1	Load output buffer from holding latch written to by AUDxDAT.
pbufld2	Like pbufld1, but only during 010→011 with attach period.

AUDxAV	Attach volume. Send data to volume latch of next channel instead of to D→A converter.
AUDxAP	Attach period. Send data to period latch of next channel instead of to the D→A converter.
penhi	Enable the high 8 bits of data to go to the D→A converter.
napnav	/AUDxAV * /AUDxAP + AUDxAV—no attach stuff or else attach volume. Condition for normal DMA and interrupt requests.
sq2,1,0	The name of the state flip-flops, MSB to LSB.

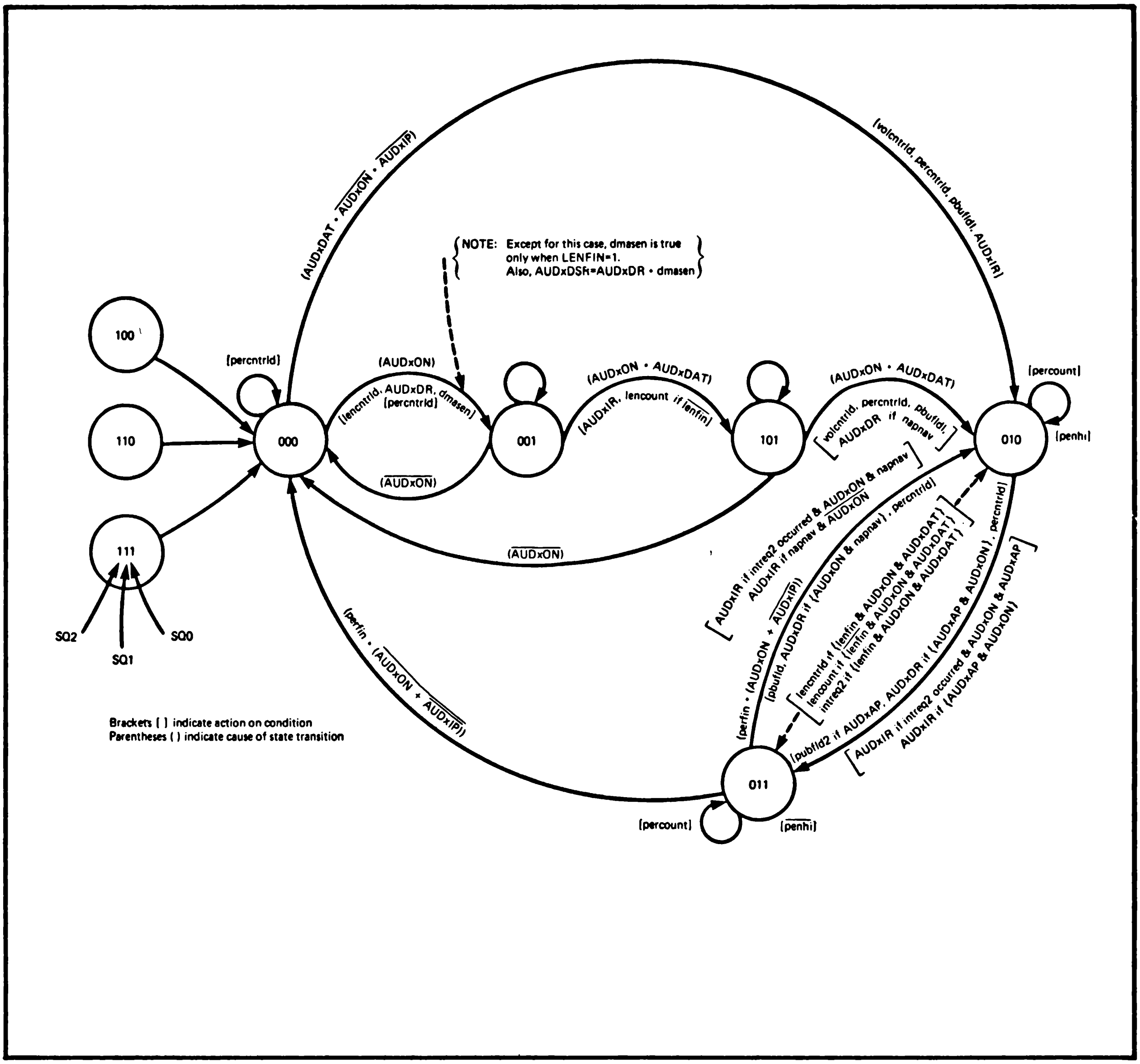

100
110
111
SQ2
SQ1
SQ0
000
001
101
010
011
[percntrld]
(AUDxON)
[lencntrld, AUDxDR, dmasen]
[percntrld]
(AUDxON)
NOTE: Except for this case, dmasen is true only when LENFIN=1. Also, AUDxDSR=AUDxDR • dmasen
(AUDxON • AUDxDAT)
[AUDxIR, lencount if lenfin]
(AUDxON • AUDxDAT)
[volcntrld, percntrld, pbufldl, AUDxDR if napnav]
(AUDxDAT • AUDxON • AUDxIP)
[volcntrld, percntrld, pbufldl, AUDxIR]
[percount]
[penhi]
(AUDxON)
[AUDxIR if intreq2 occurred & AUDxON & napnav AUDxIR if napnav & AUDxON]
(perfin • (AUDxON + AUDxIP))
[pbufld, AUDxDR if {AUDxON & napnav}, percntrld]
[lencntrld if {lenfin & AUDxON & AUDxDAT} lencount if {lenfin & AUDxON & AUDxDAT} intreq2 if {lenfin & AUDxON & AUDxDAT}]
[pubfld2 if AUDxAP, AUDxDR if {AUDxAP & AUDxON}, percntrld]
[AUDxIR if intreq2 occurred & AUDxON & AUDxAP AUDxIR if {AUDxAP & AUDxON}]
[percount]
[penhi]
(perfin • (AUDxON + AUDxIP))
Brackets [ ] indicate action on condition
Parentheses ( ) indicate cause of state transition

Figure 5-8: Audio State Diagram

Chapter 6

BLITTER HARDWARE

Introduction

The blitter is a high-performance graphics engine that uses up to four DMA channels. The operations it performs after a set-up of its registers are considerably faster than those performed by the 68000. The blitter can be used for data copying. It includes features to facilitate copying and processing of "rectangular" regions of memory. Typically, these regions are areas within graphics images. The blitter also does line drawing. The process of performing a blitter operation is sometimes called a *blit.*

The blitter uses up to four DMA channels. Three DMA channels are dedicated to retrieving data from memory to the blitter. These are designated as source A, source B, and source C. The one destination DMA channel is designated as destination D. As is shown in the following sections, it is not always necessary to use all the sources, nor is it always appropriate to use the destination DMA channel.

Each channel may be independently enabled by bits 11, 10, 9, and 8 of BLTCON0. These are called USEA, USEB, USEC, and USED. All three sources (if enabled) are fetched from memory in a pipelined fashion and held in registers for logic combination before being sent to the destination. Each channel has its own memory pointer register and its own modulo register.

A quick summary of blitter features and operations follows. Each of these topics is discussed in this chapter. The reader is also referred to the descriptions of registers whose names start with "BLT" in appendix A.

- DATA COPYING - The blitter can copy bit-plane image data anywhere in the lower 512K of memory.

- MULTIPLE POINTERS AND MODULOS - The blitter is provided with a separate pointer and modulo register for each of the sources and for the destination. This allows the blitter to move data to and from identical rectangular windows within different sizes of larger playfield images.

- ASCENDING AND DESCENDING ADDRESSING - The blitter can change addresses in an ascending or descending manner. That is, it can either start at the bottom address of both the source and the destination areas and move the data while incrementing addresses or start at the top address of the source and destination and decrement addresses during the move.

- RECTANGULAR AND LINEAR ADDRESS SCANNING - The blitter can process either linear or rectangular regions.

- LOGIC OPERATIONS - Instead of simply retrieving data from a single source, the blitter can retrieve data from up to three sources as it prepares the result for a possible destination area. Before a blit is started, the blitter is set up to perform one out of 256 possible logic operations on the three data sources as they are being transferred.

- SHIFTING - The blitter can shift one or two of its data sources up to 15 bits before applying it to the logic operation, allowing movement of images in memory across word boundaries.

- MASKING - The blitter can mask the leftmost and rightmost data word from each horizontal line. Mask registers are provided for the first and the last words on every line of blitter data. This allows logic operations on bit-boundaries from both the left and the right edge of a rectangular region.
- ZERO DETECTION - The blitter can store the result of the logic operations back into memory or simply sense whether there were any 1 bits present as a result of the logic operation. This feature can be used for hardware-assisted software collision detection.
- AREA-FILLING - The blitter can perform a hardware-assisted area fill between pre-drawn lines.
- LINE-DRAWING - The blitter can draw ordinary lines at any angle and can also apply a pattern to the lines it draws. It can also draw special lines with one pixel dot per horizontal line (a special mode needed for use with the blitter fill operation).

Data Copying

The primary purpose of the blitter is to copy (transfer) data in large blocks from one memory location to another, with or without processing. The name "blitter" stands for "block image transferrer."

Images in memory are usually stored in a linear fashion; each word of data on a line is located at an address that is one greater than the word on its left. (See figure 6-1). Note that each line is a "plus one" continuation of the previous line.

20	21	22	23	24	24	26
27	28	29	30	31	32	33
34	35	36	37	38	39	40
41	42	43	44	45	46	47
48	49	50	51	52	53	54
55	56	57	58	59	60	61

Figure 6-1: How Images are Stored in Memory

The map in figure 6-1 represents a single bit-plane (one bit of color) of an image at word addresses 20 through 61. Each of these addresses accesses one word (16 pixels) of a single bit-plane. If this image required sixteen colors, four bit-planes like this would be required in memory, and four copy (move) operations would be required to completely move the image.

The blitter is very efficient at copying such blocks because it needs to be told only the starting address (20), the destination address, and the size of the block (height = 6, width = 7). It will then automatically move the data, one word at a time, whenever the data bus is available. When the transfer is complete, the blitter will signal the processor with a flag and an interrupt.

Note that this copy (move) operation operates on memory and may or may not change the memory currently being used for display.

Pointers and Modulos

Pointer registers are used to point to the address in memory where the next word of source or destination data is located. Because pointer registers must address 512 Kbytes of memory, they occupy two 16-bit addresses. For example, the pointer for source channel A has two register addresses. BLTAPTL contains the low-order part (bits 15-0) and BLTAPTH contains the high-order part (bits 18-16) of the pointer address. Pointer registers address word boundaries so bit 0 is always a 0.

Pointer registers BLTBPTL, BLTBPTH, BLTCPTL, BLTCPTH, BLTDPTL, and BLTDPTH apply to the B, C, and D channels, respectively. The notation BLTxPTx is used to refer to the pointer registers generically.

The blitter uses modulos to allow manipulation of smaller images within larger images. A modulo is the difference between the width of the larger image and the smaller image being manipulated. There are four modulos in the blitter—BLTAMOD, BLTBMOD, BLTCMOD, and BLTDMOD. This allows each of the three sources and the destination to have a larger bit-plane image of a different size.

Modulos are 16-bit signed numbers. When they are added to the corresponding pointer register, they are sign-extended to match the larger number of bits in the pointer register. Since word addressing is used, bit 0 of the modulo is always a 0.

Figure 6-3 shows a possible bit-plane image that is larger than the source window being used by the blitter. The numbers represent the addresses (in memory) of the data words containing the image.

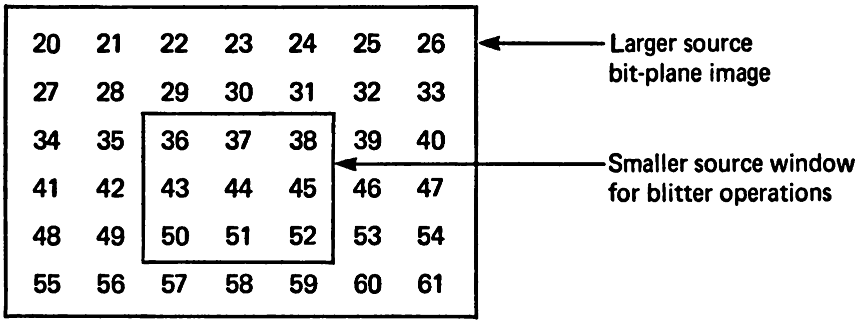

20 21 22 23 24 25 26
27 28 29 30 31 32 33
34 35 36 37 38 39 40
41 42 43 44 45 46 47
48 49 50 51 52 53 54
55 56 57 58 59 60 61
Larger source bit-plane image
Smaller source window for blitter operations

Figure 6-2: Bit-plane Image Larger than the Blitter Source Window

Note that in order to operate on the smaller window only, the address sequence must be as follows:

36, 37, 38,, 43, 44, 45,, 50, 51, 52

This requires a normal increment (+1) each time, and at the end of each window line the addition of a jump value of 4, to bring the address pointer to the start of the next window line. This jump value is called the modulo and is equal to the difference between the width of the large image and the width of the smaller window.

The blitter has a separate modulo register for each of the three possible source images and one for the destination image (four in all). This allows the larger bit map image of each source and the destination to be a different size, even though the smaller window for each is identical.

Note that although the hardware deals in words for pointers and modulos, the values loaded into these hardware registers from the 68000 are treated as byte counts. For example, a jump value of 4 for a modulo would actually be an 8 when written from the 68000.

Ascending and Descending Addressing

It is important to be able to control the direction of the address increment or decrement when the source and destination areas overlap. Ascending or descending is specified for overlapping data moves either to move a block of data or to fill a region with a particular value.

If you wish to move data toward a higher address in memory with an overlap between source and destination areas, you should use the descending (address decrement) mode for the data move. If you wish to move data toward a lower address in memory with an overlap between the source and destination areas, you should use the ascending (address increment) mode for the data move. The descending mode is selected with bit 1 of BLTCON1.

If the source and destination data areas overlap in a blitter operation, there is a possibility of writing to a particular location as the destination before it was read as the source. To prevent this kind of data destruction, you must take care to correctly choose ascending or descending mode. Also, you may need to offset the source or destination.

Using table 6-4 at the end of the chapter, you can observe the order of operations and determine the required offset or mode. Pay careful attention to the notes. It helps to draw pictures.

Rectangular or Linear Address Scanning

The BLTSIZE register is written to define the horizontal and vertical size of a rectangular region of memory. The pointer register (BLTxPTx) specifies where in memory the corresponding data block starts. The blitter adds (in ascending mode) or subtracts (in descending mode) 2 from the pointer register for each 16-bit word transferred until the count of "horizontal" words in the BLTSIZE register is met. Then it adds the contents of the modulo register (BLTxMOD) to the pointer register. The value in the modulo register thus represents the value to be added to the pointer register to get it from the point in memory just past the end of a horizontal line to the beginning of the next horizontal line of the rectangular region.

The blitter can be used to process linear rather than rectangular regions by setting the horizontal or vertical count in BLTSIZE to 1.

Blitter Logic Operations

Three sources (A, B, and C) are available to the blitter logic unit. These sources are usually one bit-plane from each of three separate graphics images. While each of these sources is a rectangular region composed of many points, the same logic operation will be performed on each point throughout the rectangular region. Accordingly, for purposes of defining the blitter logic operation it is only necessary to describe what happens for all of the possible combinations of one bit from each of the three sources. Therefore, there are only eight possible data combinations (minterms). For each of these input possibilities you need to specify whether the corresponding D (destination) output bit is on or off. This information is collected in a standard format, the LF control byte in the BLTCON0 register, shown below. This byte programs the blitter to perform one of the 256 possible logic operations on three sources for a given blit.

For example, an LF control byte of $80 (= 1000 0000 binary) turns on bits only for those points of the D destination rectangle where the corresponding bits of A, B, and C sources were all on (ABC = 1, bit 7 of LF on). All other points in the rectangle, which correspond to other combinations for A, B, and C, will be 0. This is because bits 6 through 0 of the LF control word, which specify the D output for these situations, are set to 0. The following paragraphs discuss two conceptual approaches to designing this LF control byte. One approach uses logic equations; the other uses Venn diagrams.

DESIGNING THE LF CONTROL BYTE WITH LOGIC EQUATIONS

Because it can logically combine data bits from separate image sources during a data move, the blitter is very efficient in performing graphics drawing and animation operations. For example, you could design a rectangular object to combine on-screen with a pre-existing graphic image (perhaps a car that you want to move in front of some buildings).

Producing this effect requires predrawn images of both the car and the buildings. To animate the car (that is, to move it in front of the buildings), first save the background image where the car will be placed. Next, copy the car in its first location. Then restore the old background image and save a new section of the background from the second location. Again, copy the car, this time to the second location. A continuous sequence of save, draw, and restore creates the desired effect.

Assume source A is the car image outline (mask), source B is one of the car image's bit planes, and source C is building data or background. The following operation saves the background where the car is going to be placed (destination on the left, sources on the right):

$$T = AC$$

This equation states that the background (C) should be saved (copied) to a temporary destination (T) wherever the car outline mask (A) "and" the background (C) exist together.

Now the car is placed in the background with the following operation:

$$C = AB + \overline{A}C$$

This equation states that the destination is the same as the background source (C), and background (C) should be replaced with car data (B) wherever the car outline mask (A) is true, but (or) should stay background (C) wherever the mask is not true ($\overline{A}$). Now the background must be restored (to prepare for car placement in a different location) using the following operation:

$$C = AT$$

This equation states that the background (C) should be replaced with the saved background (T) wherever the car outline mask exists (A "and" T).

If you shift the data and the mask to a new location and repeat the above three steps over and over, the car will appear to move across the background (the buildings).

Blitter Logic Operations - Combining Minterms

The blitter performs various logic operations, such as the one shown in the last section, by combining minterms. A minterm is one of eight possible logical combinations of data bits from three different data sources.

For example, the following equation uses two minterms, ABC and $AB\overline{C}$:

$$D = ABC + AB\overline{C}$$

This means that the logic value of D is a 1 if either $ABC = 1$ or $AB\overline{C} = 1$.

Another way of reading this equation is that D is true if and only if both A and B are true. This is because the equation could be grouped as:

$$D = AB\,(\,C + \overline{C}\,)$$

However, since the term $(C + \overline{C})$ is always true, this equation reduces to $D = AB$. Therefore, selecting the two minterms ABC and $AB\overline{C}$ will give the logic operation $D = AB$. These two minterms are selected with bits 7 and 6 of BLTCON0.

The minterms that can be selected by BLTCON0 control bits are as follows:

MINTERMS:	ABC	$AB\overline{C}$	$A\overline{B}C$	$A\overline{B}\overline{C}$	$\overline{A}BC$	$\overline{A}B\overline{C}$	$\overline{A}\overline{B}C$	$\overline{A}\overline{B}\overline{C}$
ENABLE BITS (BLTCON0 LF7-LF0):	7	6	5	4	3	2	1	0

Since there are eight minterms, there are 256 possible equations that can be selected.

Table of Commonly Used Equations

For your convenience, table 6-1 contains a set of commonly used equations. The last one in the table ($D = AB + \overline{A}C$) is often referred to as the "cookie-cut" minterm selector.

Table 6-1: Table of Common Minterm Values

Selected Equation	**BLTCON0 LF Code**	**Selected Equation**	**BLTCON0 LF Code**
$D = A$	F0	$D = AB$	C0
$D = \overline{A}$	0F	$D = A\overline{B}$	30
$D = B$	CC	$D = \overline{A}B$	0C
$D = \overline{B}$	33	$D = \overline{AB}$	03
$D = C$	AA	$D = BC$	88
$D = \overline{C}$	55	$D = B\overline{C}$	44
$D = AC$	A0	$D = \overline{B}C$	22
$D = A\overline{C}$	50	$D = \overline{AC}$	11
$D = \overline{A}C$	0A	$D = A + \overline{B}$	F3
$D = \overline{AC}$	05	$D = \overline{A} + \overline{B}$	3F
$D = A + B$	FC	$D = A + \overline{C}$	F5
$D = \overline{A} + B$	CF	$D = \overline{A} + \overline{C}$	5F
$D = A + C$	FA	$D = B + \overline{C}$	DD
$D = \overline{A} + C$	AF	$D = \overline{B} + \overline{C}$	77
$D = B + C$	EE	$D = AB + \overline{A}C$	CA
$D = \overline{B} + C$	BB		

Equation-to-Minterm Conversion

An example of converting an equation to minterm format in order to derive the select code is given below:

$D = AB + \overline{A}C$ (Starting equation)

$D = AB (C + \overline{C}) + \overline{A}C (B + \overline{B})$ (Multiplying by 1)

$D = ABC + AB\overline{C} + \overline{A}BC + \overline{A}\,\overline{B}C$ (Final minterms)

This final form contains only terms that contain *all* of the input sources. These are the minterms you use. These minterms are selected with the minterm enable bits LF7-LF0 as shown below:

ABC	$AB\overline{C}$	$A\overline{B}C$	$A\overline{B}\,\overline{C}$	$\overline{A}BC$	$\overline{A}B\overline{C}$	$\overline{A}\,\overline{B}C$	$\overline{A}\,\overline{B}\,\overline{C}$	(Available minterms)
1	1	0	0	1	0	1	0	(BLTCON0 LF 7-0 code in binary
C				A				LF 7-0 code in hex

DESIGNING THE LF CONTROL BYTE WITH VENN DIAGRAMS

You can use Venn diagrams as an aid in selecting minterms. The Venn diagram in figure 6-3 shows a set of three circles labeled A, B and C. In the diagram, the numbers 0 through 7 in various areas correspond to the minterm numbers shown in the preceding section.

To select which minterms are necessary to produce a certain kind of equation result, you need only examine the circles and their intersections and copy down the numbers seen there.

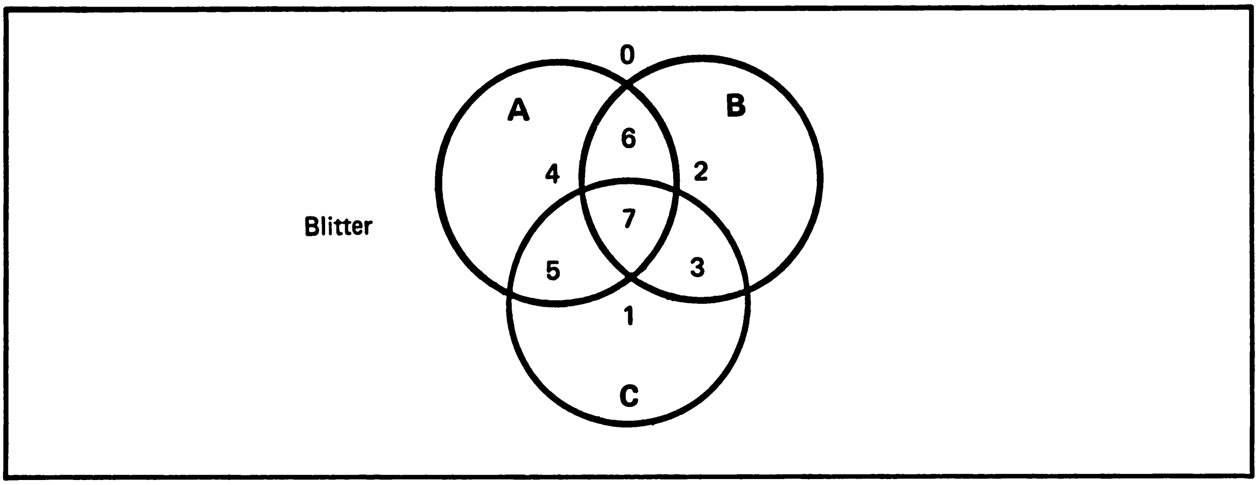


Figure 6-3: Blitter Minterm Venn Diagram

Examples of Venn Diagram Interpretation

1. If you wish to select a function D = A (that is, destination = A source only), you can select only the minterms that are totally enclosed by the A-circle in the figure above. This is the set of minterms 7, 6, 5, and 4. When written as a set of 1s for the selected minterms and 0s for those not selected, the value becomes:

7 6 5 4 3 2 1 0	MINTERM NUMBERS
1 1 1 1 0 0 0 0	SELECTED MINTERMS
F 0	equals $F0

2. If you wish to select a function that is a combination of two sources, you then look for the minterms by *both* of the circles in their common area. For example, the combination AB (A "and" B) is represented by the area common to both the A and B circles. This area encloses both minterms 7 and 6.

7 6 5 4 3 2 1 0

1 1 0 0 0 0 0 0 equals $C0.

3. If you wish to use a function that is "not" one of the sources, such as $\overline{A}$, you take all of the minterms *not* enclosed by the circle represented by A on the figure.

4. If you wish to combine minterms, you need only "or" them together. For example, the equation AB + BC results in:

```
AB =    1 1 0 0 1 0 0 0
BC =    1 0 0 0 1 0 0 0
        ---------------
        1 1 0 0 1 0 0 0 = $C8
```

Shifting

When bit-plane images are stored with sixteen 1-bit pixels in a memory word, situations arise where a particular pixel must be in a different bit position within a word before and after a block transfer.

For example, as described previously under "Logic Operations," the movement of a car image (B) across a background (C) requires both the car image (B) and the car outline mask (A) to be shifted to a new position each time the background is saved (T = AC), the car is placed ($C = AB + A\overline{C}$), and the background is restored (C = AT). As the movement proceeds, the edge of the car image can, in general, land on any bit position within a 16-bit word. This illustrates the need for a high-speed shift capability within the blitter.

Accordingly, the blitter contains a circuit known as a barrel shifter that can be used with both the A and the B data sources. It can shift these sources from 0 to 15 bits. It is a true barrel shifter; bigger shifts do not take more time than smaller shifts as they would if performed by the microprocessor. This shifter allows movement of images on pixel boundaries, even though the pixels are addressed 16 at a time by each word address of the bit-plane image.

There are two shift controls. Bits 15 through 12 of BLTCON0 select the shift value for source A. Bits 15 through 12 of BLTCON1 select the shift value for source B. Both values are normally set the same. The shift controls are used in a special way during line drawing. See "Line Drawing" below.

Masking

If an object is not an even multiple of 16 bits in width, the blitter can mask off either the left or the right edge or both in order to work with only the actual bit-boundary rectangle enclosing the object. First- and last-word masking is particularly useful when you need to store the images of a text font in a packed edge-to-edge organization.

For example, assume a packed font that contains both an "H" and an "I" as shown in figure 6-4.

```
111     1111111
 11     11  11
 11     11  11
 111111111  11
 111111111  11
 11     11  11
 11     11  11
111     1111111
```

Figure 6-4: A Packed Font

To isolate the "I" character, the first 11 bits along the left edge of the enclosing rectangle must be masked. The blitter includes this capability, called the first-word mask, and applies it to the leftmost word on each horizontal line. Only when there is a 1 bit in the first-word mask will that bit of source A actually appear in the logic operation.

For example, if the first-word mask (BLTAFWM) is 0000000000001111, the data the blitter will see, using the input for source A shown above, is shown in figure 6-5.

```
1111
 11
 11
 11
 11
 11
 11
 11
1111
```

Figure 6-5: Blitter Masking Example

In a similar way, the blitter's last-word mask (BLTALWM) masks the rightmost word of the source A data. Thus, it is possible to extract rectangular data from a source whose right and left edges are between word boundaries.

If the window is only one word wide (as illustrated above), the first and last word masks will overlap, and source-A bits will be passed only where both masks are true. This example assumed the last word mask was loaded with all 1s ($FFFF) as all masks should be when they are not needed.

Zero Detection

A blitter zero flag is provided that can be tested to determine if the logic operation selected has resulted in a null (empty = all zeros) logic operation result. The zero flag (BZERO) in bit 13 of DMACONR will stay true if the result is all zeros.

This feature is usually used to assist collision detection by "and"ing two images together to test for overlap. The operation D = AB is performed (D can actually be disabled), and if images A and B do not overlap, the zero flag will stay true.

When the purpose of a blit is only to do zero detection and not to generate a D destination image, the USED bit (bit 8 of BLTCON0) can be turned off to save time and bus cycles.

Area Filling

In addition to copying data, the blitter can simultaneously perform a fill operation during the copy. The fill operation has only one restriction: the area to be filled must be defined by first drawing untextured lines that are only one bit wide. A special line draw mode is available for this (see the "Line Drawing" section).

INCLUSIVE (NORMAL) AREA FILLING

Figure 6-6 shows a typical area fill. It demonstrates one of the bars from a bar chart.

Before	**After**
001000100	001111100
001000100	001111100
001000100	001111100
001000100	001111100

Figure 6-6: Area-fill Example — Bar Chart

A blitter line-draw is first performed to provide the two vertical lines, each one bit wide. To fill this area, you follow these steps. NOTE: A fill operation can be performed during other blitter data copy operations; however, it is often done separately, as shown here.

1. Set the modulos equal to the width of the total image minus the width of the rectangle to be filled.

   (BLTxMOD) (x = A,B,C,D)

2. Set the source and destination pointers to the same value. A case like this requires only one source and destination. This should point to the *last* (lower-right) word of the enclosing rectangle (see also item 3 below).

   (BLTxPTH, BLTxPTL) (x = A,B,C,D)

3. Run the blitter in the *descending* direction. The fill operation operates correctly only in the descending mode (right to left).

   (BLTCON1, Bit 1 = 1)

4. Use the control bit called "FCI" (for fill-carry-in) to define *how* the fill operation should be performed.

   (BLTCON1, Bit 2 = 0)

   This defines the fill start state as a 0.

5. Define the horizontal and vertical size of a rectangle of words that will enclose the lines around the area to be filled. This value must be written to the size control (BLTSIZE) register to start the fill.

The blitter uses the FCI bit as the starting fill state, beginning at the rightmost edge of each line. For each "1" bit in the source area, the blitter "flips" the fill state, either filling or not filling the space with 1's. This continues for each line until the left edge of the blit is reached. At that point, the filling stops. For another example, examine the figure below. Only the 1 bits are shown in figure 6-7. The 0 bits are blank. The figure is not drawn to scale.

```
      Before                After

1   1   1   1        11111   11111
1   1   1   1        11111   11111
 1  1    1  1         1111    1111
  1 1     1 1          111     111
   11      11           11      11
  1 1     1 1          111     111
 1  1    1  1         1111    1111
1   1   1   1        11111   11111
```

Figure 6-7: Use of the FCI Bit - Bit Is a 0

If the FCI bit is a 1 instead of a 0, the area outside the lines is filled with 1s and the area inside the lines is left with 0s in between.

Before

```
1   1     1   1
1   1     1   1
 1  1      1  1
  1 1       1 1
   11        11
  1 1       1 1
 1  1      1  1
1   1     1   1
```

After

```
111   111111    11
111   1111111   11
1111  11111111  11
11111 111111111 11
111111111111111111
11111 111111111 11
1111  11111111  11
111   1111111   11
```

Figure 6-8: Use of the FCI Bit - Bit Is a 1

EXCLUSIVE AREA FILLING

There are two fill enable bits within BLTCON1. They are called IFE (for "Inclusive Fill Enable") (used in the previous examples), and EFE (for "Exclusive Fill Enable") (used in the example below).

Exclusive fill enable means to exclude (remove) the outline on the *trailing edge* (left side) of the fill.

Since the blitter is running in descending mode during a fill, the trailing edge is formed from the leftmost of each pair of bits on a horizontal line.

If you wish to produce very sharp, single-point vertices, exclusive-fill enable must be used. Figure 6-9 shows how a single-point vertex is produced using exclusive-fill enable.

Before

```
1   1     1   1
 1  1      1  1
  1 1       1 1
   11        11
  1 1       1 1
 1  1      1  1
1   1     1   1
```

After Exclusive Fill

```
1111      1111
 111       111
  11        11
   1         1
  11        11
 111       111
1111      1111
```

Figure 6-9: Single-Point Vertex Example

Line Drawing

In addition to all the functions described above, the blitter has a line-drawing mode. The line-drawing mode is selected by placing a 1 in bit 0 of BLTCON1, which causes redefinition of some of the other control bits in BLTCON0 and BLTCON1. (See the description of the BLTCON registers in the appendix for the meanings of the other control bits.)

In line-drawing mode, the blitter has the following features:

- Draws lines up to 1,024 pixels long (twice as big as the high-resolution screen).
- Draws lines with regular or inverse video.
- Draws solid lines or textured lines.
- Draws special lines with one dot on each scan line, for use with area fill.

Many of the blitter registers serve other purposes in line-drawing mode. These registers and their functions are itemized in table 6-2 for reference purposes. Consult the appendix for more detailed descriptions of the use of these registers and control bits in line-drawing mode.

Table 6-2: Blitter Registers in Line-drawing Mode

Register Name	Bit Number	Bit Name	State	Purpose
BLTCON0	15,14,13,12	START		Code for horizontal position of first pixel
BLTCON0	11,10, 9, 8	USE	1011	Required for line-drawing
BLTCON1	15,14,13,12	BSH	0	Starts texture at bit 0
BLTCON1	5			Reserved
BLTCON1	4,3,2			Octant select code (See figure 6-10 below.)
BLTCON1	1	SING	0,1	Set for single-bit-width line
BLTCON1	0	LINE	1	Enables line-drawing mode
BLTADAT	All		8000	Index required for line-drawing
BLTBDAT	All		0 to FFFF	Line texture register
BLTSIZE	5-0	w	02	Required for line-drawing
BLTSIZE	15-6	h		Line length up to 1024
BLTAMOD	All			2(2Y - 2X) *
BLTBMOD	All			2(2Y) *
BLTCMOD	All			Width of total image
BLTDMOD	All			Width of total image
BLTAPT	All			(2Y - X) *
BLTCPT	All			Starting address of line
BLTDPT	All			Starting address of line

* Y and X are the height and width of the rectangle enclosing the line.

OCTANTS IN LINE DRAWING

Standard computer graphics texts, such as Newman and Sproul, discuss a system for dividing the Cartesian plane into eight regions called octants for purposes of line drawing. Figure 6-10 shows the numerical codes Amiga has assigned to each octant. The dotted lines in the figure represent the x-axis and y-axis.

Line drawing based on octants is a simplification that takes advantage of symmetries between x and -x, y and -y. The octant code and several values derived from delta x and delta y are loaded into blitter control registers as shown in table 6-3.

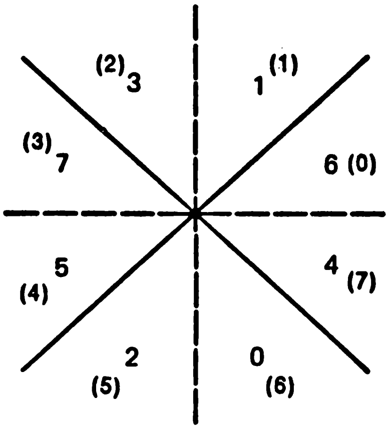


Figure 6-10: Octants for Line Drawing

In figure 6-10, the number in parentheses is the octant number and the other number stands for bits 4, 3, and 2 of register BLTCON1 as shown in table 6-3. Also see the table at the end of the description of BLTCON1 in appendix A.

Table 6-3: BLTCON1 Code Bits for Octant Line Drawing

BLTCON1 Code Bits **4 3 2**	**Octant #**
1 1 0	0
0 0 1	1
0 1 1	2
1 1 1	3
1 0 1	4
1 0 0	5
0 0 0	6
1 0 0	7

Blitter Operations and System DMA

The operations of the blitter affect the performance of the rest of the system. The following sections explain how system performance is affected by blitter direct memory access (DMA) priority, DMA time slot allocation, bus sharing between the 68000 and the bit-plane, the operations of the blitter and Copper, and different playfield display sizes.

BLITTER DMA PRIORITY

The blitter performs its various data-fetch, modify, and store operations through DMA sequences, and it shares memory access with other devices in the system. Each device that accesses memory has a priority level assigned to it, which indicates its importance relative to other devices.

Disk DMA, audio DMA, bit-plane DMA, and sprite DMA all have the highest priority level. Bit-plane DMA has priority over sprite DMA under certain circumstances. Each of these four devices is allocated a group of time slots during each horizontal scan of the video beam. If a device does not request one of its allocated time slots, the slot is open for other uses. These devices are given first priority because missed DMA cycles can cause lost data, noise in the sound output, or on-screen interruptions.

The Copper has the next priority because it has to perform its operations at the same time during each display frame to remain synchronized with the display beam sweeping across the screen.

The lowest priorities are assigned to the blitter and the 68000, in that order. The blitter is given the higher priority because it performs data copying, modifying, and line drawing operations operations much faster than the 68000.

DMA TIME SLOT ALLOCATION

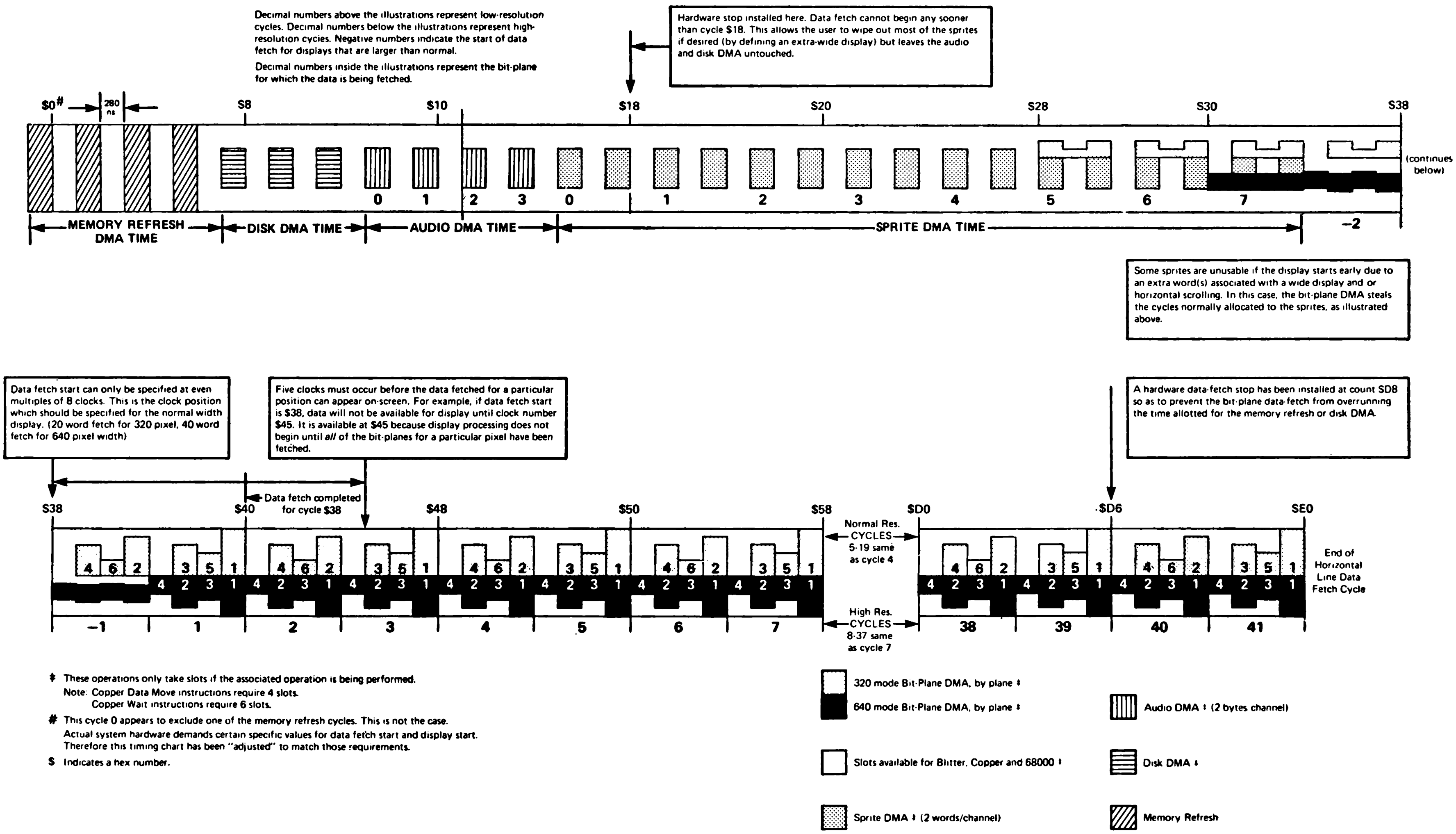
During a horizontal scan line (about 63 microseconds), there are 227.5 "color clocks", or memory access cycles. A memory cycle is approximately 280 ns in duration. The total of 227.5 cycles per horizontal line includes both display time and non-display time. Of this total time, 226 cycles are available to be allocated to the various devices that need memory access.

The time-slot allocation per horizontal line is

- 4 cycles for memory refresh (assigned to odd-numbered slots)
- 3 cycles for disk DMA
- 4 cycles for audio DMA (2 bytes per channel)
- 16 cycles for sprite DMA (2 words per channel)
- 80 cycles for bit-plane DMA (even- or odd-numbered slots according to the display size used)

Figure 6-11 shows one complete horizontal scan line and how the clock cycles are allocated.

DMA Time Slot Allocation/Horizontal line


$0#
280 ns
$8
$10
$18
$20
$28
$30
$38
0
1
2
3
0
1
2
3
4
5
6
7
−2
(continues below)
MEMORY REFRESH DMA TIME
DISK DMA TIME
AUDIO DMA TIME
SPRITE DMA TIME
$38
$40
$48
$50
$58
$D0
$D6
$E0
4
6
2
3
5
1
4
2
3
1
−1
1
2
3
4
5
6
7
38
39
40
41
Data fetch completed for cycle $38
Normal Res. CYCLES 5-19 same as cycle 4
High Res. CYCLES 8-37 same as cycle 7
End of Horizontal Line Data Fetch Cycle
320 mode Bit-Plane DMA, by plane ‡
640 mode Bit-Plane DMA, by plane ‡
Slots available for Blitter, Copper and 68000 ‡
Sprite DMA ‡ (2 words/channel)
Audio DMA ‡ (2 bytes channel)
Disk DMA ‡
Memory Refresh

Decimal numbers above the illustrations represent low-resolution cycles. Decimal numbers below the illustrations represent high-resolution cycles. Negative numbers indicate the start of data fetch for displays that are larger than normal.

Decimal numbers inside the illustrations represent the bit-plane for which the data is being fetched.

Hardware stop installed here. Data fetch cannot begin any sooner than cycle $18. This allows the user to wipe out most of the sprites if desired (by defining an extra-wide display) but leaves the audio and disk DMA untouched.

Some sprites are unusable if the display starts early due to an extra word(s) associated with a wide display and or horizontal scrolling. In this case, the bit-plane DMA steals the cycles normally allocated to the sprites, as illustrated above.

Data fetch start can only be specified at even multiples of 8 clocks. This is the clock position which should be specified for the normal width display. (20 word fetch for 320 pixel, 40 word fetch for 640 pixel width)

Five clocks must occur before the data fetched for a particular position can appear on-screen. For example, if data fetch start is $38, data will not be available for display until clock number $45. It is available at $45 because display processing does not begin until *all* of the bit-planes for a particular pixel have been fetched.

A hardware data-fetch stop has been installed at count $D8 so as to prevent the bit-plane data-fetch from overrunning the time allotted for the memory refresh or disk DMA.

‡ These operations only take slots if the associated operation is being performed.
Note: Copper Data Move instructions require 4 slots.
Copper Wait instructions require 6 slots.

# This cycle 0 appears to exclude one of the memory refresh cycles. This is not the case.
Actual system hardware demands certain specific values for data fetch start and display start.
Therefore this timing chart has been "adjusted" to match those requirements.

$ Indicates a hex number.

Figure 6-11: DMA Time Slot Allocation

BIT-PLANE/PROCESSOR BUS SHARING

The 68000 uses only the even-numbered memory access cycles. The 68000 spends about half of a complete processor instruction time doing internal operations and the other half accessing memory. Therefore, the allocation of alternate memory cycles to the 68000 makes it appear to the 68000 that it has the memory all of the time, and it will run at full speed.

Some 68000 instructions do not match perfectly with the allocation of even cycles and cause cycles to be missed. If cycles are missed, the 68000 must wait until its next available memory slot before continuing. However, most instructions do not cause cycles to be missed, so the 68000 runs at full speed most of the time if there is no blitter DMA interference.

Figure 6-12 illustrates the normal cycle of the 68000.

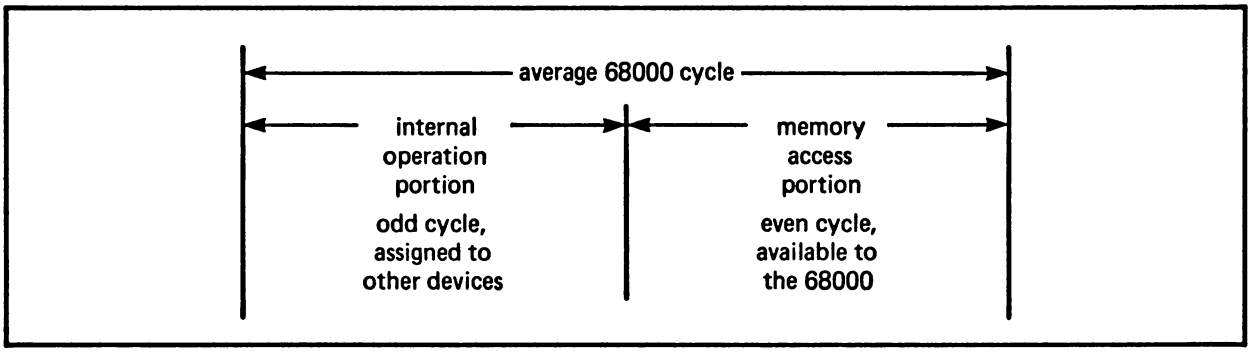

average 68000 cycle
internal operation portion
odd cycle, assigned to other devices
memory access portion
even cycle, available to the 68000

Figure 6-12: Normal 68000 Cycle

If the display contains four or fewer low-resolution bit-planes, the 68000 can be granted alternate memory cycles (if it is ready to ask for the cycle and is the highest priority item at the time). However, if there are more than four bit-planes, bit-plane DMA will begin to steal cycles from the 68000 during the display.

During the display time for a six-bit-plane display (low resolution, 320 pixels wide), 160 time slots will be taken by bit-plane DMA for each horizontal line. As you can see from figure 6-13, bit-plane DMA steals 50 percent of the open slots that the processor might have used if there were only four bit-planes displayed.

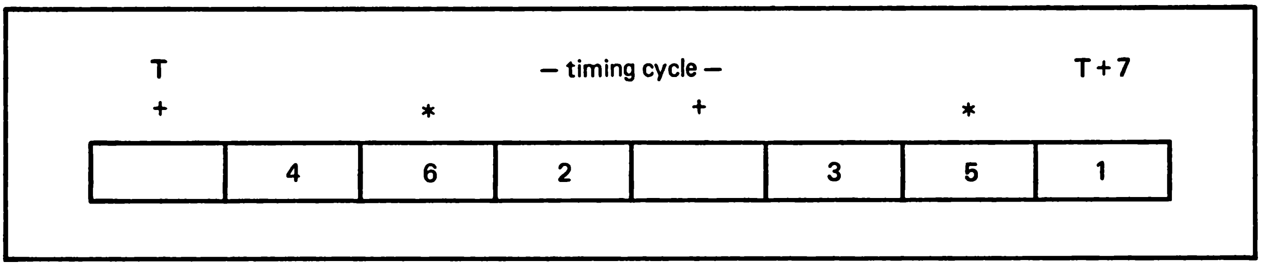


Figure 6-13: Time Slots Used by a Six-bit-plane Display

Notes for figure 6-13:

+ an open memory slot that the 68000 might use

* a slot that cannot be used by the 68000 because of added bit-plane DMA

If you specify four high-resolution bit-planes (640 pixels wide), bit-plane DMA needs *all* of the available memory time slots during the display time just to fetch the 40 data words for each line of the four bit-planes (40 x 4 = 160 time slots). This effectively locks out the 68000 (as well as the blitter or Copper) from any memory access during the display.

Figure 6-14 shows how the time slots are allocated for high-resolution bit-planes.

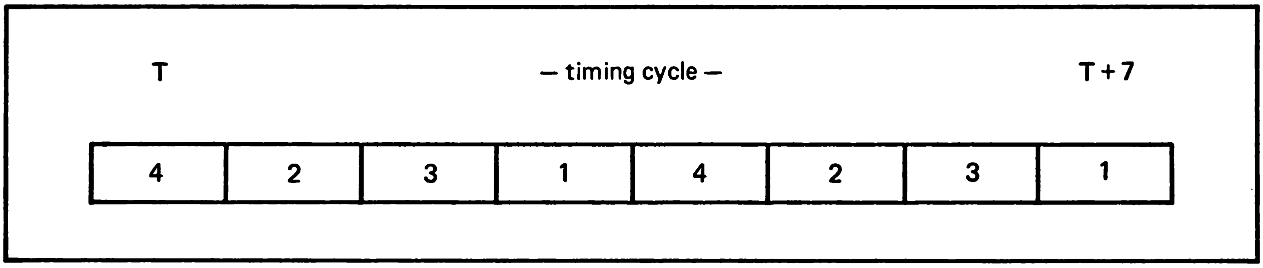


Figure 6-14: Time Slots Used by a High-resolution Display

EFFECTS OF DIFFERENT DISPLAY SIZES

Each horizontal line in a normal, full-sized display contains 320 pixels in low-resolution mode or 640 pixels in high-resolution mode. Thus, either 20 or 40 words will be fetched during the horizontal line display time. If you want to scroll a playfield, one extra data word per line must be fetched from the memory.

Display size is adjustable (see chapter 3, "Playfield Hardware"), and bit-plane DMA takes precedence over sprite DMA. As shown in figure 6-11, larger displays may block out one or more of the highest-numbered sprites, especially with scrolling.

EFFECTS OF BLITTER OPERATION

As mentioned above, the blitter normally has a higher priority than the processor for DMA cycles. There are certain cases, however, when the blitter and the 68000 can share memory cycles. If given the chance, the blitter would steal every available memory cycle. Display, disk, and audio DMA take precedence over the blitter, so it cannot block them from bus access. Depending on the setting of the blitter DMA mode bit, commonly referred to as the "blitter-nasty" bit, the processor may be blocked from bus access. This bit is called BLTPRI (for "blitter has priority over processor") and is in register DMACONW.

If BLTPRI is a 1, the blitter will keep the bus for every available memory cycle. This could potentially be every cycle.

If BLTPRI is a 0, the DMA manager will monitor the 68000 cycle requests. If the 68000 is unsatisfied for three consecutive memory cycles, the blitter will release the bus for one cycle.

Table 6-4 shows all of the possible operating modes of the blitter, along with the distribution of its memory access windows. The table shows three words of a blit (the first word, any middle word, and the last word) and how bus activity occurs for this sequence. The following conventions are used in this table:

- A, B, and C stand for the sources.
- D stands for the destination.
- Numerical suffixes indicate which word within a blit is being fetched. For example A0 is the first memory word fetch; A1 is any middle memory word fetch; and A2 is the last memory word fetch.

Table 6-4: Typical Blitter Cycle Sequence

USE Code in BLTCON0	Active Channels				Cycle Sequence												
F	A	B	C	D	A0	B0	C0	-	A1	B1	C1	D0	A2	B2	C2	D1	D2
E	A	B	C		A0	B0	C0	A1	B1	C1	A2	B2	C2				
D	A	B		D	A0	B0	-	A1	B1	D0	A2	B2	D1	-	D2		
C	A	B			A0	B0	-	A1	B1	-	A2	B2					
B	A		C	D	A0	C0	-	A1	C1	D0	A2	C2	D1	-	D2		
A	A		C		A0	C0	A1	C1	A2	C2							
9	A			D	A0	-	A1	D0	A2	D1	-	D2					
8	A				A0	-	A1	-	A2								
7		B	C	D	B0	C0	-	-	B1	C1	D0	-	B2	C2	D1	-	D2
6		B	C		B0	C0	-	B1	C1	-	B2	C2					
5		B		D	B0	-	-	B1	D0	-	B2	D1	-	D2			
4		B			B0	-	-	B1	-	-	B2						
3			C	D	C0	-	-	C1	D0	-	C2	D1	-	D2			
2			C		C0	-	C1	-	C2								
1				D	D0	-	D1	-	D2								
0	none				-	-	-	-									

Notes for table 6-4:

- No fill.
- No competing bus activity.
- Three-word blit.
- Typical operation involves fetching all sources twice before the first destination becomes available. This is due to internal pipelining. Care must be taken with overlapping source and destination regions.

Table 6-4 is only meant to be an illustration of the typical order of blitter cycles on the bus. Bus cycles are dynamically allocated based on blitter operating mode; competing bus activity from processor, bit-planes, and other DMA channels; and other factors. Commodore-Amiga does not guarantee the accuracy of or future adherence to this chart. We reserve the right to make product improvements or design changes in this area without notice.

Complete Blitter Example

The following example sets up the blitter to clear a block of memory. This program assumes you have the required include files to get correct magic numbers.

This code is meant to be only an example. Programmers who wish to use the blitter directly and who want their code to perform with the rest of the Amiga software must do the appropriate **OwnBlitter()**, **DisownBlitter()**, and **WaitBlit()** calls. See the *Amiga ROM Kernel Manual* for information about using these calls.

```
    include 'exec/types.i'
    include      'hardware/custom.i'
    include 'hardware/blit.i'

    xref         _custom
;
;Busy-wait for the previous blit to complete
;
WAITBLIT:
    BTST     #DMAB_BLTDONE-8,DMACONR(A1)
    BNE.S    WAITBLIT
    RTS
;
;This routine uses a side affect in the blitter. When each of the blits is
;finished, the pointer in the blitter is pointing to the next word to be blitted.
;A0 = pointer to first word to clear
;0 = number of even bytes to clear
;
CLEARMEM:
    LEA      _CUSTOM,A1                  ;Get pointer to chip registers
    MOVE.L   A0,BLTPTD(A1)               ;Set up destination to clear
    CLR.W    BLTMDD(A1)                  ;Set modulo to no-skip
    ASR.L    #1,D0                       ;Convert to number of words
    CLR.W    BLTCON1(A1)                 ;No special modes
    MOVE.W   #DEST,BLTCON0(A1)           ;Minterms = 0, enable only destination
;This routine splits the blit into several parts to feed the blitter.
;First, the leftovers.
    MOVEQ    #$3F,D1
    AND.W    D0,D1                       ;Extract non-64-words-at-a-time part
    BEQ.S    LABEL1                      ;Even up the blit with a small one first
             SUB.L    D1,D0
             OR       #$40,D1            ;Make it one row X leftover words
             MOVE.W   D1,BLTSIZE(A1)     ;Trigger the blit
```

```
LABEL1:
;                                Note:  the upper word of d1 is already zero
    MOVE.W        #$FFC0,D1                             ;Now look at more upper bits
    AND.W         D0,D1                                 ;Extract 10 more bits
    BEQ.S         LABEL2                                ;Any to do?
                  SUB.L          D1,D0                  ;How many 128-Kbyte blocks left
                  BSR            WAITBLIT               ;Wait for any previous blit to complete
                  MOVE.W         D0,BLTSIZE(A1)         ;Trigger next blit
LABEL2:
    SWAP          D0
    BEQ.S         DONE                                  ;Check for any bits set in upper word
                  CLR.W          D1                     ;Will do blits 128 Kbytes at a time
LOOP:
    BSR           WAITBLIT
;                                Need move for this to work on 68000
    MOVE          D1,BLTSIZE(A1)                        ;Trigger a big blit
                  SUBQ.W         #1,D0                  ;Could be a dbf
                  BNE.S          LOOP                   ;Any more 128-Kbyte blits?
DONE:
;                                Exit.  Note:  blit may still be in progress.
;                                The support to manage async blits is one of the
;                                reasons to use the system software from Amiga.
    RTS
```

Blitter Block Diagram

Figure 6-15 shows the basic building blocks for a single bit of a 16-bit-wide operation of the blitter. It does not cover the line-drawing hardware.

1. The upper left corner shows how the first- and last-word masks are applied to the incoming A-source data. When the blit shrinks to one word wide, *both* the first- and last-word masks apply to the incoming data word.

2. The shifter (upper right and center left) drawing illustrates how 16 bits of data is taken from a specified position within a 32-bit register, based on the A-shift or B-shift values shown in BPLCON0 and BPLCON1.

3. The minterm generator (center right) illustrates how the minterm select bits either allow or inhibit the use of a specific minterm.

4. The drawing shows how the fill operation works on the data generated by the minterm combinations. Fill operations can be performed simultaneously with other complex logic operations.

5. At the bottom, the drawing shows that data generated for the destination can be prevented from being written to a destination by using one of the blitter control bits.

6. Not shown on this diagram is the logic for zero detection, which looks at every bit generated for the destination. If there are *any* 1-bits generated, this logic indicates that the area of the blit contained at least one 1-bit (zero detect is false).

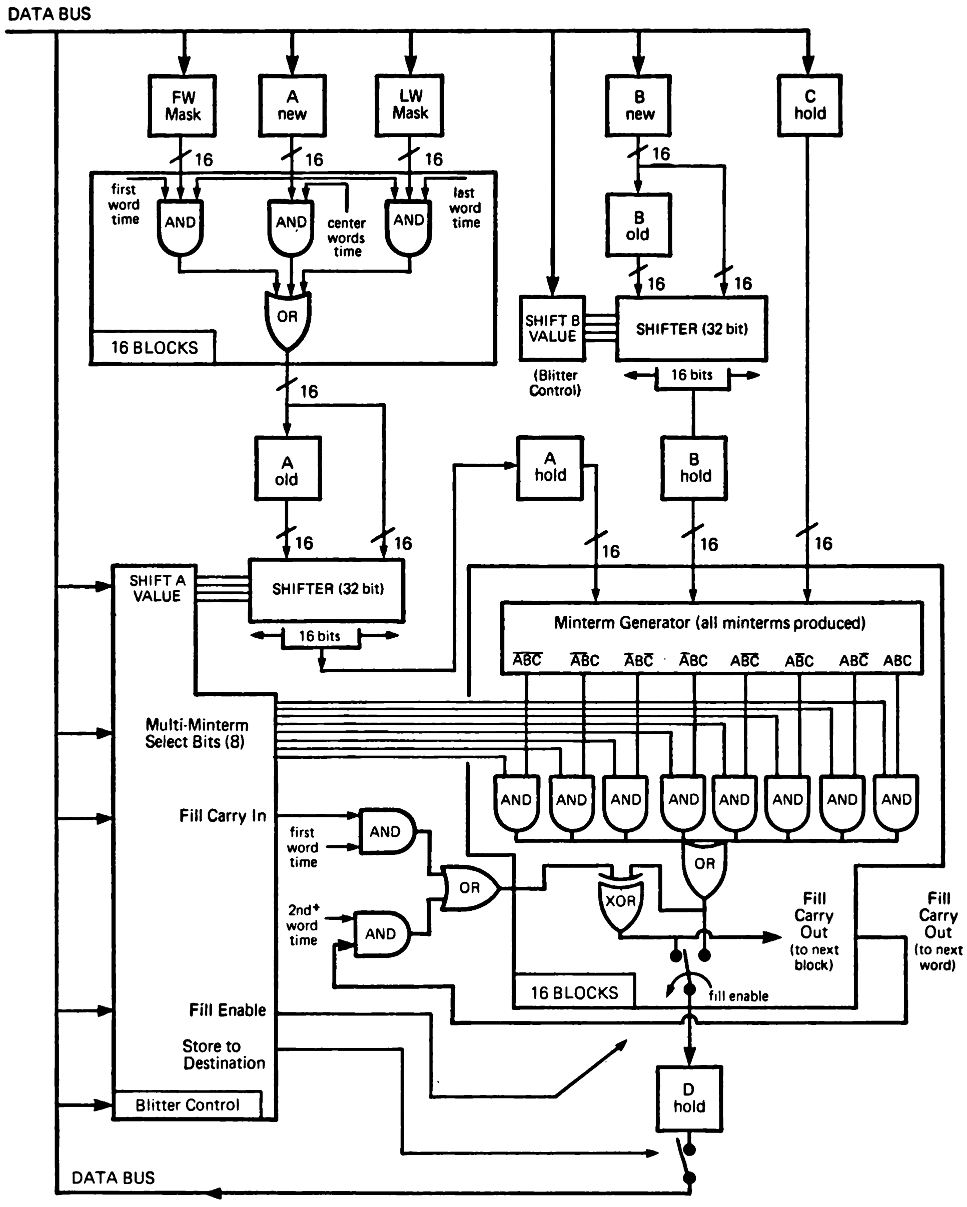

DATA BUS
FW Mask
A new
LW Mask
B new
C hold
16
16
16
16
first word time
AND
AND
center words time
AND
last word time
OR
16 BLOCKS
16
B old
16
16
SHIFT B VALUE
SHIFTER (32 bit)
(Blitter Control)
16 bits
A old
A hold
B hold
16
16
16
16
16
SHIFT A VALUE
SHIFTER (32 bit)
16 bits
Minterm Generator (all minterms produced)
$\overline{A}\overline{B}\overline{C}$
$\overline{A}\overline{B}C$
$\overline{A}B\overline{C}$
$\overline{A}BC$
$A\overline{B}\overline{C}$
$A\overline{B}C$
$AB\overline{C}$
ABC
Multi-Minterm Select Bits (8)
AND
AND
AND
AND
AND
AND
AND
AND
Fill Carry In
first word time
AND
OR
2nd+ word time
AND
OR
XOR
Fill Carry Out (to next block)
Fill Carry Out (to next word)
16 BLOCKS
fill enable
Fill Enable
Store to Destination
Blitter Control
D hold
DATA BUS

Figure 6-15: Blitter Block Diagram

Chapter 7

SYSTEM CONTROL HARDWARE

Introduction

This chapter covers the control hardware of the Amiga system, including the following topics:

- How playfield priorities may be specified relative to the sprites

- How collisions between objects are sensed
- How system direct memory access (DMA) is controlled
- How interrupts are controlled and sensed

Video Priorities

You can control the priorities of various objects on the screen to give the illusion of three dimensions. The section below shows how playfield priority may be changed relative to sprites.

FIXED SPRITE PRIORITIES

You cannot change the relative priorities of the sprites. They will always appear on the screen with the lower-numbered sprites appearing in front of (having higher screen priority than) the higher-numbered sprites. This is shown in figure 7-1. Each box represents the image of the sprite number shown in that box.

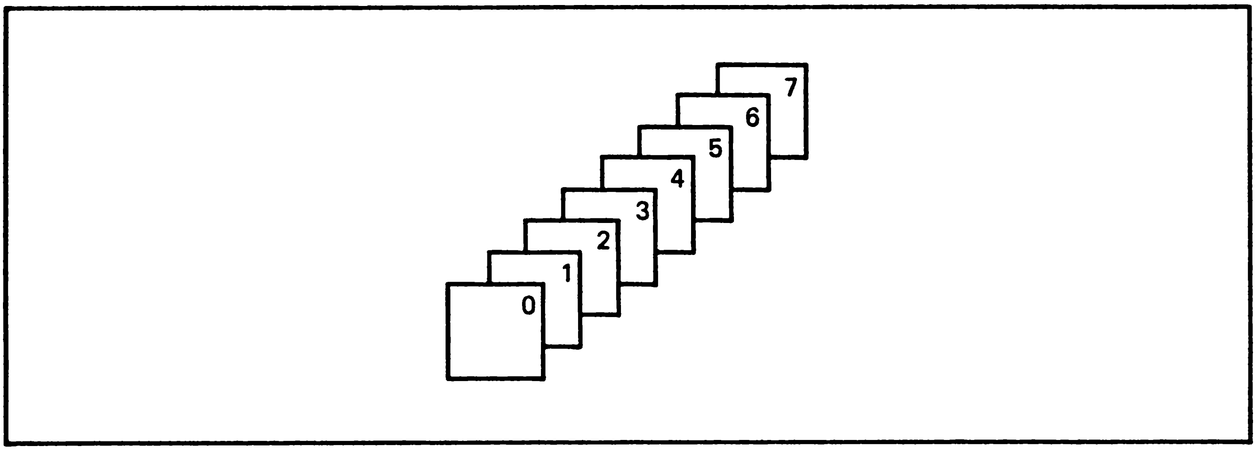


Figure 7-1: Inter-Sprite Fixed Priorities

HOW SPRITES ARE GROUPED

For playfield priority and collision purposes only, sprites are treated as four groups of two sprites each. The groups of sprites are:

Sprites 0 and 1
Sprites 2 and 3
Sprites 4 and 5
Sprites 6 and 7

UNDERSTANDING VIDEO PRIORITIES

The concept of video priorities is easy to understand if you imagine that four fingers of one of your hands represent the four pairs of sprites and two fingers of your other hand represent the two playfields. Just as you cannot change the sequence of the four fingers on the one hand, neither can you change the relative priority of the sprites. However, just as you can intertwine the two fingers of one hand in many different ways relative to the four fingers of the other hand, so can you position the playfields in front of or behind the sprites. This is illustrated in figure 7-2.

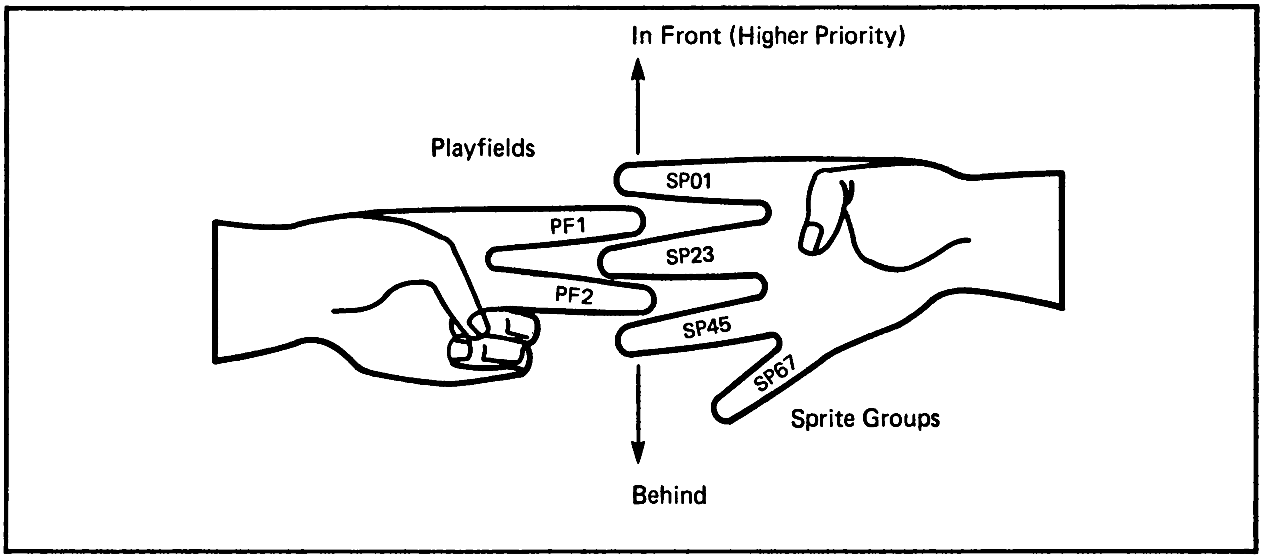


Figure 7-2: Analogy for Video Priority

Five possible positions can be chosen for each of the two "playfield fingers." For example, you can place playfield 1 on top of sprites 0 and 1 (0), between sprites 0 and 1 and sprites 2 and 3 (1), between sprites 2 and 3 and sprites 4 and 5 (2), between sprites 4 and 5 and sprites 6 and 7 (3), or beneath sprites 6 and 7 (4). You have the same possibilities for playfield 2.

The numbers 0 through 4 shown in parentheses in the preceding paragraph are the actual values you use to select the playfield priority positions. See "Setting the Priority Control Register" below.

You can also control the priority of playfield 2 relative to playfield 1. This gives you additional choices for the way you can design the screen priorities.

SETTING THE PRIORITY CONTROL REGISTER

This register lets you define how objects will pass in front of each other or hide behind each other. Normally, playfield 1 appears in front of playfield 2. The PF2PRI bit reverses this relationship, making playfield 2 more important. You control the video priorities by using the bits in BPLCON2 (for "bit-plane control register number 2") as shown in table 7-1.

Table 7-1: Bits in BPLCON2

Bit Number	Name	Function
15-7		Not used (keep at 0)
6	PF2PRI	Playfield 2 priority
5-3	PF2P2 - PF2P0	Playfield 2 placement with respect to the sprites
2-0	PF1P2 - PF1P0	Playfield 1 placement with respect to the sprites

The binary values that you give to bits PF1P2-PF1P0 determine where playfield 1 occurs in the priority chain as shown in table 7-2. This matches the description given in the previous section.

Table 7-2: Priority of Playfields Based on Values of Bits PF1P2-PF1P0

Value	**Placement (from most important to least important)**				
000	PF1	SP01	SP23	SP45	SP67
001	SP01	PF1	SP23	SP45	SP67
010	SP01	SP23	PF1	SP45	SP67
011	SP01	SP23	SP45	PF1	SP67
100	SP01	SP23	SP45	SP67	PF1

In this table, PF1 stands for playfield 1, and SP01 stands for the group of sprites numbered 0 and 1. SP23 stands for sprites 2 and 3 as a group; SP45 stands for sprites 4 and 5 as a group; and SP67 stands for sprites 6 and 7 as a group.

Bits PF2P2-PF2P0 let you position playfield 2 among the sprite priorities in exactly the same way. However, it is the PF2PRI bit that determines which of the two playfields appears in front of the other on the screen. Here is a sample of possible BPLCON2 register contents that would create something a little unusual:

BITS	15-7	PF2PRI	PF2P2-0	PF1P2-0
VALUE	0s	1	010	000

This will result in a sprite/playfield priority placement of:

PF1 SP01 SP23 PF2 SP45 SP67

In other words, where objects pass across each other, playfield 1 is in front of sprite 0 or 1; and sprites 0 through 3 are in front of playfield 2. However, playfield 2 is in front of playfield 1 in any area where they overlap and where playfield 2 is not blocked by sprites 0 through 3.

Collision Detection

You can use the hardware to detect collisions between one sprite group and another sprite group, any sprite group and either of the playfields, the two playfields, or any combination of these items.

The first kind of collision is typically used in a game operation to determine if a missile has collided with a moving player. The second kind of collision is typically used to keep a moving object within specified on-screen boundaries. The third kind of collision detection allows you to define sections of playfield as individual objects, which you may move using the blitter. This is called playfield animation. If one playfield is defined as the backdrop or playing area and the other playfield is used to define objects (in addition to the sprites), you can sense collisions between the playfield-objects and the sprites or between the playfield-objects and the other playfield.

HOW COLLISIONS ARE DETERMINED

The video output is formed when the input data from all of the bit-planes and the sprites is combined into a common data stream for the display. For each of the pixel positions on the screen, the color of the highest priority object is displayed. Collisions are detected when two or more objects attempt to overlap in the same pixel position. This will set a bit in the collision data register.

HOW TO INTERPRET THE COLLISION DATA

The collision data register, CLXDAT, is *read-only*, and its contents are automatically cleared to 0 after it is read. Its bits are as shown in table 7-3.

Table 7-3: CLXDAT Bits

Bit Number	Collisions Registered
15	not used
14	Sprite 4 (or 5) to sprite 6 (or 7)
13	Sprite 2 (or 3) to sprite 6 (or 7)
12	Sprite 2 (or 3) to sprite 4 (or 5)
11	Sprite 0 (or 1) to sprite 6 (or 7)
10	Sprite 0 (or 1) to sprite 4 (or 5)
9	Sprite 0 (or 1) to sprite 2 (or 3)
8	Even bit-planes to sprite 6 (or 7)
7	Even bit-planes to sprite 4 (or 5)
6	Even bit-planes to sprite 2 (or 3)
5	Even bit-planes to sprite 0 (or 1)
4	Odd bit-planes to sprite 6 (or 7)
3	Odd bit-planes to sprite 4 (or 5)
2	Odd bit-planes to sprite 2 (or 3)
1	Odd bit-planes to sprite 0 (or 1)
0	Even bit-planes to odd bit-planes

The notes in parentheses in table 7-3 refer to collisions that will register only if you want them to show up. The collision control register described below lets you either ignore or include the odd-numbered sprites in the collision detection.

Notice that in this table, collision detection does *not* change when you select either single- or dual-playfield mode. Collision detection depends only on the actual bits present in the odd-numbered or even-numbered bit-planes. The collision control register specifies how to handle the bit-planes during collision detect.

HOW COLLISION DETECTION IS CONTROLLED

The collision control register, CLXCON, contains the bits that define certain characteristics of collision detection. Its bits are shown in table 7-4.

Table 7-4: CLXCON Bits

Bit Number	Name	Function
15	ENSP7	Enable sprite 7 (OR with sprite 6)
14	ENSP5	Enable sprite 5 (OR with sprite 4)
13	ENSP3	Enable sprite 3 (OR with sprite 2)
12	ENSP1	Enable sprite 1 (OR with sprite 0)
11	ENBP6	Enable bit-plane 6 (match required for collision)
10	ENBP5	Enable bit-plane 5 (match required for collision)
9	ENBP4	Enable bit-plane 4 (match required for collision)
8	ENBP3	Enable bit-plane 3 (match required for collision)
7	ENBP2	Enable bit-plane 2 (match required for collision)
6	ENBP1	Enable bit-plane 1 (match required for collision)
5	MVBP6	Match value for bit-plane 6 collision
4	MVBP5	Match value for bit-plane 5 collision
3	MVBP4	Match value for bit-plane 4 collision
2	MVBP3	Match value for bit-plane 3 collision
1	MVBP2	Match value for bit-plane 2 collision
0	MVBP1	Match value for bit-plane 1 collision

Bits 15-12 let you specify that collisions with a sprite pair are to include the odd-numbered sprite of a pair of sprites. The even-numbered sprites always are included in the collision detection. Bits 11-6 let you specify whether to include or exclude specific bit-planes from the collision detection. Bits 5-0 let you specify the polarity (true-false condition) of bits that will cause a collision. For example, you may wish to register collisions only when the object collides with "something green" or "something blue." This feature, along with the collision enable bits, allows you to specify the exact bits, and their polarity, for the collision to be registered.

NOTES

This register is *write-only*. If all bit-planes are excluded (disabled), then a bit-plane collision will *always* be detected.

Beam Position Detection

Sometimes you might want to synchronize the 68000 processor to the video beam that is creating the screen display. In some cases, you may also wish to update a part of the display memory *after* the system has already accessed the data from the memory for the display area.

The address for accessing the beam counter is provided so that you can determine the value of the video beam counter and perform certain operations based on the beam position. Note, however, that the Copper is already capable of watching the display position for you and doing certain register-based operations automatically. Refer to "Copper Interrupts" below and chapter 2, "Coprocessor Hardware," for further information.

In addition, when you are using a light pen with this system, this same address is used to read the light pen position rather than the beam position. This is described fully in chapter 8, "Interface Hardware."

USING THE BEAM POSITION COUNTER

There are four addresses that access the beam position counter. Their usage is described in table 7-5.

Table 7-5: Contents of the Beam Position Counter

VPOSR	*Read-only*	Read the high bit of the vertical position (V8) and the frame-type bit.
	Bit 15	LOF (Long-frame bit). Used to initialize interlaced displays.
	Bits 14-1	Unused
	Bit 0	High bit of the vertical position (V8). Allows PAL line counts (313) to appear in PAL versions of the Amiga.
VHPOSR	*Read-only*	Read vertical and horizontal position of the counter that is producing the beam on the screen (also reads the light pen).
	Bits 15-8	Low bits of the vertical position, bits V7-V0
	Bits 7-0	The horizontal position, bits H8-H1. Horizontal resolution is 1/160th of the screen width.
VPOSW	*Write only*	Bits same as VPOSR above.
VHPOSW	*Write only*	Bits same as VHPOSR above. Used for counter synchronization with chip test patterns.

As usual, the address pairs VPOSR,VHPOSR and VPOSW,VHPOSW can be read from and written to as long words, with the most significant addresses being VPOSR and VPOSW.

Interrupts

This system supports the full range of 68000 processor interrupts. The various kinds of interrupts generated by the hardware are brought into the peripherals chip and are translated into six of the seven available interrupts of the 68000.

NONMASKABLE INTERRUPT

Interrupt level 7 is the nonmaskable interrupt and is not generated anywhere in the current system. The raw interrupt lines of the 68000, IPL2 through IPL0, are brought out to the expansion connector and can be used to generate this level 7 interrupt for debugging purposes.

MASKABLE INTERRUPTS

Interrupt levels 1 through 6 are generated. Control registers within the peripherals chip allow you to mask certain of these sources and prevent them from generating a 68000 interrupt.

USER INTERFACE TO THE INTERRUPT SYSTEM

The system software has been designed to correctly handle all system hardware interrupts at levels 1 through 6. A separate set of input lines, designated INT2* and INT6* [1] have been routed to the expansion connector for use by external hardware for interrupts. These are known as the external low- and external high-level interrupts.

These interrupt lines are connected to the peripherals chip and create interrupt levels 2 and 6, respectively. It is recommended that you take advantage of the interrupt handlers built into the operating system by using these external interrupt lines rather than generating interrupts directly on the processor interrupt lines.

---

[1] A * indicates an active low signal.

INTERRUPT CONTROL REGISTERS

There are two interrupt registers, interrupt enable (mask) and interrupt request (status). Each register has both a read and a write address.

The names of the interrupt addresses are

INTENA
Interrupt enable (mask) - *write only.* Sets or clears specific bits of INTENA.

INTENAR
Interrupt enable (mask) read - *read only.* Reads contents of INTENA.

INTREQ
Interrupt request (status) - *write only.* Used by the processor to force a certain kind of interrupt to be processed (software interrupt). Also used to clear interrupt request flags once the interrupt process is completed.

INTREQR
Interrupt request (status) read - *read only.* Contains the bits that define which items are requesting interrupt service.

The bit positions in the interrupt request register correspond directly to those same positions in the interrupt enable register. The only difference between the read-only and the write-only addresses shown above is that bit 15 has no meaning in the read-only addresses.

SETTING AND CLEARING BITS

Below are the meanings of the bits in the interrupt control registers and how you use them.

Set and Clear

The interrupt registers, as well as the DMA control register, use a special way of selecting which of the bits are to be set or cleared. Bit 15 of these registers is called the SETCLR bit.

When you wish to *set* a bit (make it a 1), you must place a 1 in the position you want to set *and a 1 into position 15.*

When you wish to *clear* a bit (make it a 0), you must place a 1 in the position you wish to clear *and a 0 into position 15.*

Positions 14-0 are bit-selectors. You write a 1 to any one or more bits to *select* that bit. At the same time you write a 1 or 0 to bit 15 to either *set* or *clear* the bits you have selected. Positions 14-0 that have 0 value will *not* be affected when you do the write. If you want to set some bits and clear others, you will have to write this register twice (once for setting some bits, once for clearing others).

Master Interrupt Enable

Bit 14 of the interrupt registers (INTEN) is for interrupt enable. This is the master interrupt enable bit. If this bit is a 0, it disables *all* other interrupts. You may wish to clear this bit to temporarily disable all interrupts to do some critical processing task.

NOTE

> This bit is used for enable/disable only. It creates no interrupt request.

External Interrupts

Bits 13 and 3 of the interrupt registers are reserved for external interrupts.

Bit 13, EXTER, becomes a 1 when the system line called INT6* becomes a logic 0. Bit 13 generates a level 6 interrupt.

Bit 3, PORTS, becomes a 1 when the system line called INT2* becomes a logic 0.

Bit 3 causes a level 2 interrupt.

Vertical Blanking Interrupt

Bit 5, VERTB, causes an interrupt at line 0 (start of vertical blank) of the video display frame. The system is often required to perform many different tasks during the vertical blanking interval. Among these tasks are the updating of various pointer registers, rewriting lists of Copper tasks when necessary, and other system-control operations.

The minimum time of vertical blanking is 20 horizontal scan lines (begins at line 0 and ends at line 20). You also have control over where (after line 20) the display actually starts by using the DIWSTRT (display window start) register (see chapter 3, "Playfield Hardware"). This can extend the effective vertical blanking time.

If you find that you still require additional time during vertical blanking, you can use the Copper to create a level 3 interrupt. This Copper interrupt would be timed to occur just after the last line of display on the screen (after the display window stop which you have defined by using the DIWSTOP register).

Copper Interrupt

Bit 4, COPER, is used by the Copper to issue a level 3 interrupt. The Copper can change the content of *any* of the bits of this register, as it can write any value into most of the machine registers. However, this bit has been reserved for specifically identifying the Copper as the interrupt source.

Generally, you use this bit when you want to sense that the display beam has reached a specific position on the screen, and you wish to change something in memory based on this occurrence.

Audio Interrupts

Bits 10 - 7, AUD3 - 0, are assigned to the audio channels. They are called AUD3, AUD2, AUD1, and AUD0 and are assigned to channels 3, 2, 1, and 0, respectively.

This level 4 interrupt signals "audio block done." When the audio DMA is operating in automatic mode, this interrupt occurs when the last word in an audio data stream has been accessed. In manual mode, it occurs when the audio data register is ready to accept another word of data.

See chapter 5, "Audio Hardware," for more information about interrupt generation and timing.

Blitter Interrupt

Bit 6, BLIT, signals "blitter finished." If this bit is a 1, it indicates that the blitter has completed the requested data transfer. The blitter is now ready to accept another task.

This bit generates a level 3 interrupt.

Disk Interrupt

Bits 12 and 1 of the interrupt registers are assigned to disk interrupts.

Bit 12, DSKSYN, indicates that the sync register matches disk data. This bit generates a level 5 interrupt

Bit 1, DSKBLK, indicates "disk block finished." It is used to indicate that the specified disk DMA task that you have requested has been completed. This bit generates a level 1 interrupt.

More information about disk data transfer and interrupts may be found in chapter 8, "Interface Hardware."

Serial Port Interrupts

The following serial interrupts are associated with the specified bits of the interrupt registers.

Bit 11, RBF (for receive buffer full), specifies that the input buffer of the UART has data that is ready to read. This bit generates a level 5 interrupt.

Bit 0, TBE (for "transmit buffer empty"), specifies that the output buffer of the UART needs more data and data can now be written into this buffer. This bit generates a level 1 interrupt.

DMA Control

Many different direct memory access (DMA) functions occur during system operation. There is a read address as well as a write address to the DMA register so you can tell which DMA channels are enabled.

The address names for the DMA register are as follows:

DMACONR - Direct Memory Access Control - *read-only.*

DMACON - Direct Memory Access Control - *write-only.*

The contents of this register are shown in table 7-5 (bit on if enabled).

Table 7-6: Contents of DMA Register

Bit Number	Name	Function
15	SET/CLR	The set/reset control bit. See description of bit 15 under "Interrupts" above.
14	BBUSY	Blitter busy status - *read-only*
13	BZERO	Blitter zero status - *read-only.* Remains 1 if, during a blitter operation, the blitter output was always zero.
12, 11		Unassigned
10	BLTPRI	Blitter priority. Also known as "blitter-nasty." When this is a 1, the blitter has full (instead of partial) priority over the 68000.
9	DMAEN	DMA enable. This is a master DMA enable bit. It enables the DMA for all of the channels at bits 8-0.
8	BPLEN	Bit-plane DMA enable
7	COPEN	Coprocessor DMA enable
6	BLTEN	Blitter DMA enable
5	SPREN	Sprite DMA enable
4	DSKEN	Disk DMA enable
3-0	AUDxEN	Audio DMA enable for channels 3-0 (x = 3 - 0).

For more information on using the DMA, see the following chapters:

Sprites - chapter 4, "Sprite Hardware"

Bit-planes - chapter 3, "Playfield Hardware"

Blitter - chapter 6, "Blitter Hardware"

Disk - chapter 8, "Interface Hardware"

Audio - chapter 5, "Audio Hardware"

Copper - chapter 2, "Coprocessor Hardware"

Chapter 8

INTERFACE HARDWARE

Introduction

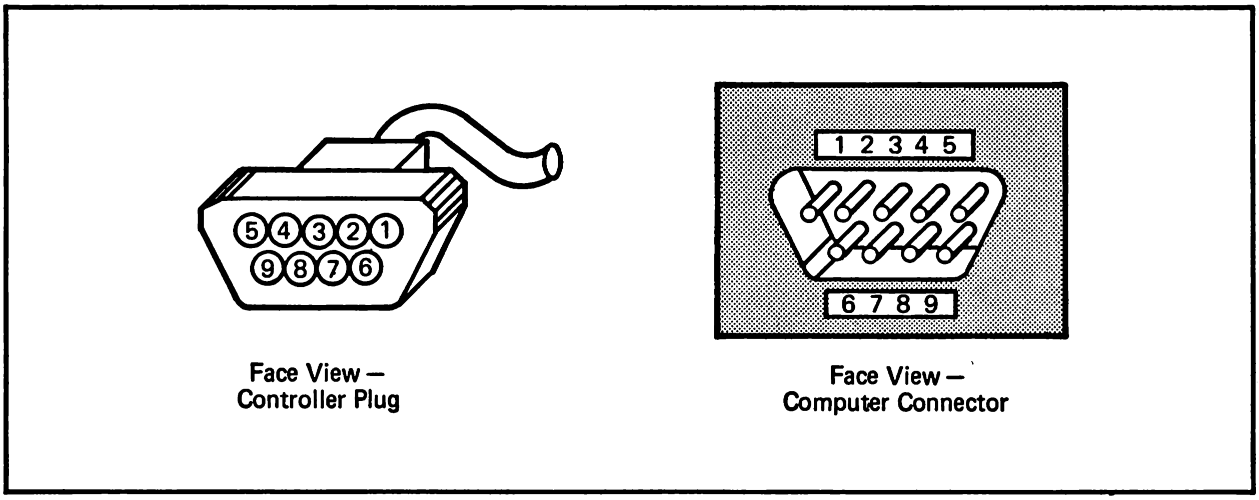
This chapter covers the ways in which the Amiga talks to the outside world, including the following features:

- Mouse/joystick/light pen ports

- Disk controller
- Keyboard
- Parallel I/O interface
- RS-232-C compatible serial interface (for external modems or serial devices)
- RAM cartridge slot (for expansion to 512K bytes)
- Expansion bus interface
- Audio output jacks
- Video output connectors (RGB, NTSC, RF modulator)

Controller Port Interface

On the side of the computer, there are two nine-pin connectors that can be used for many different kinds of controllers. Figure 8-1 shows one of the two computer connectors and the corresponding face-on view of the typical controller plug.



Face View —
Controller Plug

Face View —
Computer Connector

Figure 8-1: Controller Plug and Computer Connector

READING THE CONTROLLER PORT

Mouse controllers, joysticks, proportional controllers, and light pens use the same connector, but they sometimes have considerably different functions. Therefore, the pins function differently depending on the type of controller used.

Mouse/Trackball Controllers

The inputs for the mouse or trackball are the same as those for the joystick switches in these ways:

- The joystick "right" and "back" switches are the same as the pins used for mouse or trackball horizontal motion detection.
- The joystick "left" and "forward" switches are the same as the pins used for mouse or trackball vertical motion detection.

Pulses enter these inputs from the mouse or trackball and are converted into an up count or a down count when motion occurs. In the following discussion only the mouse action is described; the trackball activity is identical.

Direction of Motion versus Count

Imagine that the mouse is being moved on the table over an exact image of the screen itself. The movements of the on-screen object controlled by the mouse correspond exactly to the movements the user makes with the mouse itself (all directions of movement are exactly the same).

The counter counts up when the mouse is moved to the right or "down" (toward you). The counter counts down when the mouse is moved to the left or "up" (away from you).

The coordinates X,Y indicate the controlled object's position on the screen. The coordinates X=0, Y=0 are at the upper left-hand corner of the screen, and the coordinates X=Xmax, Y=Ymax are at the lower right-hand corner.

Reading the Counters

The mouse/trackball counter contents can be accessed by reading register addresses named JOY0DAT and JOY1DAT. These contain the counts for the left (0) and the right (1) controller ports.

The contents of each of these 16-bit registers are as follows:

Bits 15-8	Mouse/trackball vertical count
Bits 7-0	Mouse/trackball horizontal count

Counter Limitations

These counters will "wrap around" in either the positive or negative direction. If you wish to use the mouse to control something that is happening on the screen, you must read the counters once each vertical blanking period and save the previous contents of the registers. Then you can subtract to determine direction of movement and speed.

The counter contents must be read once each vertical blanking time to find out if the user moved the mouse since counters were last read.

The mouse produces about 200 count pulses per inch of movement in either a horizontal or vertical direction. Vertical blanking happens once each 1/60th of a second. If you read the mouse once each vertical blanking period, you will most likely find a count difference (from the previous count) of less than 127. (Only if a user moves the mouse at a speed of more than 72 inches per second will it exceed this count—an unlikely happening).

If you subtract the current count from the previous count, the absolute value of the difference will represent the speed. The sign of the difference (positive or negative), along with the sign of the previous and current values, lets you determine which direction the mouse is traveling.

The example shown in table 8-1 treats both counts as unsigned values, ranging from 0 to 255. A count of 100 pulses is measured in each case.

Table 8-1: Determining the Direction of the Mouse

Previous Count	**Current Count**	**Direction**
200	100	Up (Left)
100	200	Down (Right)
200	45	Down *
45	200	Up **

Notes for table 8-1:

* Because 200-45 = 155, which is more than 127, the true count must be 255 - (200-45) = 100; and the direction is down.

** 45-200 = -155. Because the absolute value of -155 exceeds 127, the true count must be 255 + (-155) = 100; and the direction is up.

There are two buttons on the Amiga mouse. However, the control circuitry supports mice and trackballs with as many as three buttons if desired.

o Button 1 (left button on Amiga mouse) is connected to pin 6 of the controller port. Trigger-lines are read for each of the controller ports by reading PA7 (port 1 fire button) or PA6 (port 0 fire button) of the odd-addressed 8520 peripheral ports. A logic state of 1 means "switch open." A logic state of 0 means "switch closed."

o Button 2 (right button on Amiga mouse) is connected to pin 9 of the controller ports. It is read as one of the potentiometer ports. See "Reading Proportional Controllers" for more information. High resistance indicates "switch open." Low resistance indicates "switch closed."

o Button 3, when used, is connected as the other proportional controller input. This is pin 5 of the controller ports.

Joystick Controllers

The joystick controllers have four simple direction switches and one trigger button. The direction switches are connected to pins 1, 2, 3, and 4 as FORWARD, BACK, LEFT, and RIGHT. The trigger button is connected to pin 6.

The normal state of each of the switches is open. This places a logic 1 on each of the input lines. When a switch is closed, it is connected to ground (pin 8), placing a logic 0 on the line.

Reading the joystick input data logic states is not so simple, however, because the data registers for the joysticks are the same as the counters that are used for the mouse or trackball controllers. These are named JOY0DAT (port 0) and JOY1DAT (port 1).

Table 8-2 shows how to interpret the data once you have read it from these registers. The true logic state of the switch data in these registers is "1 = switch closed."

Table 8-2: Interpreting Data from JOY0DAT and JOY1DAT

Data Bit	**Interpretation**
1	True logic state of "right" switch.
9	True logic state of "left" switch .
1 (XOR) 0	You must calculate the exclusive-or of bits 1 and 0 to obtain the logic state of the "back" switch.
9 (XOR) 8	You must calculate the exclusive-or of bits 9 and 8 to obtain the logic state of the "forward" switch.

Proportional Controllers

Each of the game controller ports can handle two variable-resistance input devices, also known as proportional input devices. This section describes how the positions of the proportional input devices can be determined. There are two common types of proportional controllers: the "paddle" controller pair and the X-Y proportional joystick. A paddle controller pair consists of two individual enclosures, each containing a single resistor and fire-button and each connected to a common controller port input connector. The typical connection is as shown in figure 8-2.

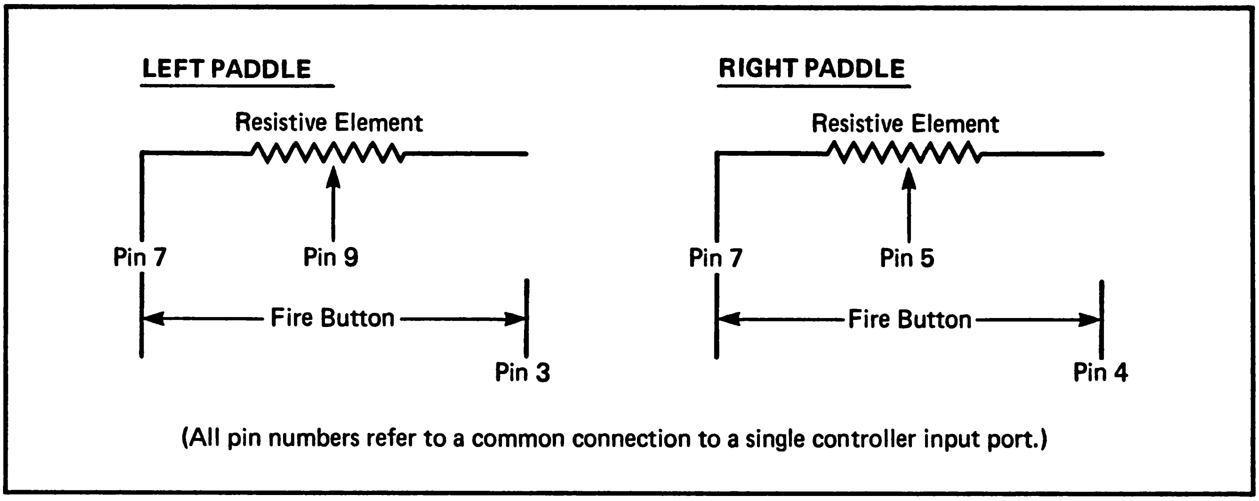

LEFT PADDLE
Resistive Element
Pin 7
Pin 9
Fire Button
Pin 3
RIGHT PADDLE
Resistive Element
Pin 7
Pin 5
Fire Button
Pin 4
(All pin numbers refer to a common connection to a single controller input port.)

Figure 8-2: Typical Paddle Controller Connection

In an X-Y proportional joystick, the resistive elements are connected individually to the X and Y axes of a single controller enclosure (instead of being in separate enclosures). Typical connections are shown in table 8-3.

Table 8-3: Typical Controller Connections

Pin	Joystick	Mouse, Trackball, Driving Controller	Proportional Controller (Pair)	X-Y Variable Proportional Joystick
1	Forward*	V-pulse	(unused)	(unused)
2	Back*	H-pulse	(unused)	(unused)
3	Left*	VQ-pulse	Left button	Button 1
4	Right*	HQ-pulse	Right button	Button 2
5	(unused)	Button (3) (if used)	Right POT	POT X
6	Button(1)	Button(1)	(unused)	(unused)
7	+5V	+5V	+5V	+5V
8	GND	GND	GND	GND
9	Button(2) (if used)	Button(2)	Left POT	POT Y

An asterisk (*) at the end of a name indicates active low.

Reading Proportional Controller Buttons

For the two-control paddle controllers, the left and right joystick inputs serve as the fire buttons for the left and right controllers.

Interpreting Proportional Controller Position

Interpreting the position of the controller requires some preliminary work. This is an activity normally done during the vertical blanking interval (and is part of the operating system function).

During vertical blanking, you write a value into an address called POTGO. For a standard X-Y joystick, this value is hex 0001. Writing to this register starts the operation of some special hardware that reads the potentiometer values and sets the values contained in the POT registers (described below) to zero.

The read circuitry stays in a reset state for the first seven or eight horizontal video scan lines. Following the reset interval, the circuit allows a charge to begin building up on a timing capacitor whose charge rate will be controlled by the position of the external controller resistance. For each horizontal scan line thereafter, the circuit compares the charge on the timing capacitor to a preset value. If the charge is below that value, one count is added to the counter for that POT. If it is above that value, the counter value will be held at the stopped value until the next POTGO is issued.

Effects of Different Resistance on Charging Rate

You normally issue POTGO at the beginning of a video screen, then read the values in the POT registers during the next vertical blanking period, just before issuing POTGO again. (Again note that this is an automatic feature of the operating system.)

Nothing in the system prevents the counters from overflowing (wrapping past a count of 255). However, the system is designed to insure that the counter cannot overflow within the span of a single screen. This allows you to know for certain whether an overflow is indicated by the controller.

Although there are 262 or 263 possible horizontal scan lines on a single NTSC video screen, each of the POT counters is eight bits wide, which allows a maximum of 255 in any counter. This is why the control circuitry is delayed seven or eight horizontal scan lines—to limit the maximum POT count value to 255.

Proportional Controller Registers

The following registers are used for the proportional controllers:

POT0DAT - port 0 data (vertical/horizontal)
POT1DAT - port 1 data (vertical/horizontal)

Bit positions:

Bits 15-8	POT0Y value or POT1Y value
Bits 7-0	POT0X value or POT1X value

All counts are reset to zero when POTGO is written. Counts are normally read one screen after the scan circuitry is enabled.

Potentiometer Specifications

The resistance of the potentiometers should be a linear taper. Based on the design of the integrating analog-to-digital converter used, the maximum resistance should be no more than 528K (470K +/- 10 percent is suggested) for either the X or Y pots. This is based on a charge capacitor of 0.047uf, +/- 10 percent, and a maximum time of 16.6 milliseconds for charge to full value (one video frame time).

Light Pen

A light pen can be connected only to the left controller port (port 0). Its connections are not shown in table 8-3. The pins controller port pins normally used by a light pen are shown in table 8-4.

Table 8-4: Light Pen Pin Usage

Pin Number	Usage
7	+5V
8	GND
5	Pen-pressed-to-screen
6	Capture beam position

The signal called "pen-pressed-to-screen" is generally a single switch to ground, normally open, which is actuated by a switch in the nose of the light pen. Note that this switch is connected to one of the potentiometer inputs and must be treated as such. The signal called "capture beam position" is connected as the trigger switch of a normal joystick.

The principles of light pen operation are as follows (assuming the light pen has been enabled):

1. Just as the system exits vertical blank, the capture circuitry for the light pen is automatically enabled.

2. The video beam starts to create the picture, sweeping from left to right for each horizontal line as it paints the picture from the top of the screen to the bottom.

3. The light pen creates a trigger signal at the moment that the video beam passes the window in the nose of the pen.

4. This trigger signal tells the internal circuitry to capture and save the current contents of the beam register, VPOSR. This allows you to determine where the pen was placed by reading the exact horizontal and vertical value of the counter beam at the instant the beam passed the light pen.

Reading the Light Pen Registers

The light pen register is at the same address as the beam counter, VPOSR and VHPOSR. The bits are as follows:

VPOSR:	Bit 15	Long frame
	Bits 14-1	Unused
	Bit 0	V8 (most significant bit of vertical position)
VHPOSR:	Bits 15-8	V7-V0 (vertical position)
	Bits 7-0	H8-H1 (horizontal position)

The software can refer to this register set as a long word whose address is VPOSR.

The positional resolution of these registers is as follows:

Vertical	1 scan line in non-interlaced mode 2 scan lines in interlaced mode
Horizontal	2 low-resolution pixels in either high- or low-resolution

To enable the light pen input, write a 1 into bit 3 of BPLCON0 (bit-plane control register 0). Once the light pen input is enabled and the light pen issues a trigger signal, the value in VPOSR is frozen. (The counters still count; only the read value is frozen.) This freeze is released at the end of internal vertical blanking (vertical position 20). No single bit in the system can tell you that the light pen has been triggered, but it can be determined as follows:

1. Read (long) VPOSR twice.

2. If both values are not the same, the light pen has not triggered since the last top-of-screen (V = 20).

3. If both values are the same, mask off the upper 15 bits of the 32-bit word and compare it with the hex value of \$10500 (V=261).
4. If the VPOSR value is greater than \$10500, the light pen has not triggered since the last top-of-screen. If the value is less, the light pen has triggered and the value read is the screen position of the light pen.

A somewhat simplified method of determining the truth of the light pen value involves instructing the system software to read the register *only* during the internal vertical blanking period of $0 < V20$:

1. Read (long) VPOSR once, during the period of $0 < V20$.
2. Mask off the upper 15 bits of the 32-bit word and compare it with the hex value of \$10500 (V=261).
3. If the VPOSR value is greater than \$10500, the light pen has not triggered since the last top-of-screen. If the value is less, the light pen has triggered and the value read is the screen position of the light pen.

Adapting to Special Controllers

The Amiga can read and interpret controllers other than the standard joystick or proportional controller by using the control lines built into the POTGO register (address DFF034) to redefine the functions of some of the controller port pins.

Table 8-5 is a copy of part of the POTGO register bit description, paraphrased from appendix A of this manual. POTGO (DFF034) is the write-only address for the pot control register. POTGOR (DFF016) is the read-only address for the pot control register. The pot-control register controls a four-bit bidirectional I/O port that shares the same four pins as the four pot inputs.

Table 8-5: POTGO Register

Bit Number	Name	Function
15	OUTRY	Output enable for right port pin 9
14	DATRY	I/O data right port pin 9
13	OUTRX	Output enable for right port pin 5
12	DATRX	I/O data right port pin 5
11	OUTLY	Output enable for left port pin 9
10	DATLY	I/O data left port pin 9
09	OUTLX	Output enable for left port pin 5
08	DATLX	I/O data left port pin 5
07-01	X	Reserved for chip identification
00	START	Start pots (dump capacitors, start counters)

Instead of using the pot pins as variable-resistive inputs, you can use these pins as a four-bit input/output port. This provides you with the equivalent of two additional pins on each of the two controller ports for general purpose I/O, as shown in table 8-5.

If you set any of the "OUT..." bits high, it disconnects the potentiometer control circuitry from the port. The current state of the "DAT..." pins in this register—1 or 0—will appear on the specified port pin. You set the state of the OUT... and DAT... pins by writing into this register through the POTGO address. There are large capacitors on these lines, and it can take up to 300 microseconds for the line to change state.

To use this register as an input, sensing the current state of the pot pins, write all 0s to POTGO. Thereafter you can read the current state by using read-only address POTGOR. Any bits set as inputs will be affected by the START bit of the POTGO register. You can also use these signals for fire-buttons. To do this, drive the line high (set both OUT... and DAT... to 1). When the button is pressed, the line will be shorted to ground, and reading POTGOR will produce a 0. If the button is not pressed, the reading will be 1.

Disk Controller

The disk controller in this system can handle four double-sided, 3 1/2- or 5 1/4-inch disk drives. A 3 1/2-inch drive is installed in the basic unit. The other drives are external to the main box.

Control of the disk operations is distributed among several registers in the system Among the control types are

- Selection, motor control, sensing
- Disk DMA channel control, DMA enable.
- Disk data read/write.
- Disk format control.
- Interrupts generated.

DISK SELECTION, CONTROL, AND SENSING

The disk subsystem uses two 8520 ports plus one FLAGS interrupt port. The specific bits used are detailed in table 8-6.

CIA A ($BFE001), port A, has four input bits allocated to the disk subsystem. CIA B ($BFD000), port B, is entirely dedicated to output bits for the disk.

Table 8-6· Disk Subsystem

Port	Pin	Name	Function
CIA A	PA5	DSKRDY*	Disk ready (active low).
CIA A	PA4	DSKTRACK0*	Disk heads currently positioned over track zero (active low).
CIA A	PA3	DSKPROT*	Disk is write protected (active low).
CIA A	PA2	DSKCHANGE*	Disk has been removed from the drive. The drives that support this signal latch it until the next time the heads are stepped (active low).

CIA B	PB7	DSKMOTOR*	Disk motor control (active low). This signal is nonstandard on the Amiga system. Each drive will latch the motor signal at the time its SELn* signal turns on. The disk drive motor will stay in this state until the next time SELn* turns on, at which time it will latch the new value of DSKMOTOR*. All software that selects drives should set up the motor signal before selecting any drives. The drive will "remember" the state of its motor when it is not selected. All drive motors turn off after system reset.
CIA B	PB6	DSKSEL3*	Select drive 3 (active low).
CIA B	PB5	DSKSEL2*	Select drive 2 (active low).
CIA B	PB4	DSKSEL1*	Select drive 1 (active low).
CIA B	PB3	DSKSEL0*	Select drive 0 (internal drive) (active low).
CIA B	PB2	DSKSIDE*	Specify a particular head of the disk. Zero implies the upper head.
CIA B	PB1	DSKDIREC	Specify the direction to seek the heads. Zero implies seek towards the center spindle. Track zero is at the outside of the disk.
CIA B	PB0	DSKSTEP*	Step the heads of the disk. This signal should always be used as a pulse (first low, then high). Leaving this line low while changing the SEL lines confuses the change logic
CIA B	FLAG	DSKINDEX*	Disk index pulse (BFDD00, bit 4). Can be used to create level 6 interrupt

Disk DMA Channel Control

Data is normally transferred to the disk by direct memory access (DMA). The disk DMA is controlled by four items:

- Pointer to the area into which or from which the data is to be moved
- Length of data to be moved by DMA
- Direction of data transfer (read/write)
- DMA enable

Pointer to Data

You specify the 19-bit-wide byte address from which or to which the data is to be transferred. The lowest bit (bit 0) of this address is treated as a 0. (You cannot start data on an odd-byte boundary.)

This address must be written into registers named DSKPTH and DSKPTL. DSKPTH gets the high three bits of the pointer, DSKPTL gets the low sixteen bits of the pointer. These registers are positioned at two consecutive word addresses on a long word boundary within the register space. This allows you to initialize both registers by a single write of a long word to the address of DSKPTH.

Length, Direction, DMA Enable

All of the control bits relating to this topic are contained in a single register, called DSKLEN. Its bits are shown in table 8-7.

Table 8-7: DSKLEN Register

Bit Number	Name	Usage
15	DMAEN	Disk DMA enable
14	WRITE	Disk write (RAM → disk if 1)
13-0	LENGTH	Number of words to transfer

The bit called DMAEN and the system DMA control bit for the disk must be set in order to allow disk DMA to occur. See chapter 7, "System Control Hardware," for more information about system DMA controls.

The hardware requires a special sequence in order to start DMA to the disk. This sequence prevents accidental writes to the disk. In short, the DMAEN bit in the DSKLEN register must be turned on twice in order to actually enable the disk DMA hardware. Here is the sequence you should follow:

1. Set this register to $4000, thereby forcing the DMA for the disk to be turned off.
2. Put the value you want into the DSKLEN register.
3. Write this value again into the DSKLEN register. This actually starts the DMA.
4. After the DMA is complete, set the DSKLEN register back to $4000, to prevent accidental writes to the disk.

As each data word is transferred, the LENGTH value is decremented. As each transfer occurs, the value of the pointer DSKPTH, DSKPTL is incremented. This points to the area where the next word of data will be written or read. When the LENGTH value counts down to 0, the transfer stops.

The recommended method of reading from the disk is to read an entire track into a buffer and then search for the sector(s) that you want. With this process you need to read from the disk only once for the entire track. In addition, there are no time-critical sections in reading this way, so that other high-priority subsystems (such as graphics or audio, both of which have stringent real time constraints) are allowed to run.

If you have too little memory for track buffering (or for some other reason decide not to read a whole track at once), the disk hardware supports a limited set of sector-searching facilities. There is a register that may be polled to examine the disk input stream.

There is a hardware bug that causes the last three bits of data sent to the disk to be lost. Also, the last word in a disk-read DMA operation may not come in (that is, one less word may be read than you asked for).

OTHER REGISTERS IN DISK OPERATIONS

A number of other registers are also associated with disk operations, as specified below.

DSKBYTR - Disk Data Byte and Status Read

This register is the disk-microprocessor data buffer. In read mode, data from the disk is loaded into this register one byte at a time. As each byte is received into the register, the BYTEREADY bit is set true. BYTEREADY is cleared each time the DSKBYTR register is read.

DSKBYTR is the register normally used by system software to synchronize the processor to the disk rotation before issuing a read or write under DMA control. The bits are shown in table 8-8.

Table 8-8: DSKBYTR Register

Bit Number	Name	Function
15	BYTEREADY	Indicates that this register contains a valid byte of data (reset by reading this register).
14	DMAON	The DMA bit (in DSKLEN) is enabled and the DMACON bits are on, too. All DMA bits must be on for this to be true.
13	DISKWRITE	This disk write bit (in DSKLEN) is enabled.
12	WORDEQUAL	Indicates the DISKSYNC register equals the disk input stream. This bit is true only while the input stream matches the sync register (as little as two microseconds).
11-8	Unused	
7-0	DATA	Disk byte data.

ADKCON and ADKCONR - Audio and Disk Control Register

ADKCON is the write address and ADKCONR is the read address for this register. The bottom eight bits of the register are used for the audio circuitry. The other bits are shown in table 8-9.

Table 8-9: ADKCON and ADKCONR Register

Bit Number	Name	Function
15	CLR/SET	Same use as in the DMA enable register. Bit 15 must be a 1 if the register bits are to be set. Bit 15 is a 0 if the bits are to be cleared.
14	PRECOMP1	MSB of Precomp specifier
13	PRECOMP0	LSB of Precomp specifier Value of 00 selects none. Value of 01 selects 140 ns. Value of 10 selects 280 ns. Value of 11 selects 560 ns.
12	MFMPREC	Value of 0 selects GCR Precomp. Value of 1 selects MFM Precomp.
11	UARTBRK	Value of 1 forces the output of the Paula special chip's serial port to 0 (an RS-232-C break).
10	WORDSYNC	Value of 1 enables synchronizing and starting of DMA on disk read of a word. The word on which to synchronize must be written into the DSKSYNC address (DFF07E).
9	MSBSYNC	Value of 1 enables sync on MSBit (GCR).
8	FAST	Value of 1 selects two microseconds per bit cell (usually MFM), 0 selects four microseconds per bit (usually GCR).
7-4	ATPER3-0	Audio attach-period controls (not disk-related).
3-0	ATVOL3-0	Audio attach-volume controls (not disk-related).

One form of GCR format is the format used by the Apple[tm] computer. Data bytes on Apple-formatted disks always have the most significant bit set to a 1. When reading a GCR formatted disk, the software must use a translate table called a nibble-izer to assure that all data written to the disk conforms with this bit-setting. Bit 9, when a 1, tells the disk controller to look for this sync bit on every disk byte.

DSKSYNC - Disk Input Synchronizer

The DSKSYNC register is used to synchronize the input stream. If WORDEQUAL is enabled in ADKCON, no data is transferred to memory until a word is found in the input stream that matches the word in the DSKSYNC register. In addition, the DSKSYNC bit in INTREQ is set when the input stream matches the DSKSYNC register. The DSKSYNC bit in INTREQ is independent of the WORDEQUAL enable.

DSKDAT and DSKDATR Disk DMA Data Registers

These register addresses are for testing purposes only.

DSKDAT is write-only and DISKDATR is a read-only, early-read dummy address. This register is the disk DMA data buffer. It contains two bytes of data that are either sent (written) to or received (read) from the disk. The write mode is enabled by bit 14 of the DSKLEN register. The DMA controller automatically transfers data to or from this register and RAM.

DISK INTERRUPTS

The disk controller can issue two kinds of interrupts:

- DSKSYNC (level 5, INTREQ bit 12)—the input stream matches the DSKSYNC register.
- DSKBLK (level 1, INTREQ bit 1)—disk DMA has completed.

Each of these is explained further in the sections titled "Length, Direction, DMA Enable" and "Other Registers Involved with Disk Operations." See chapter 7, "System Control Hardware," for more information about interrupts.

The Keyboard

The keyboard is interfaced to the system through one pair of lines connected to the odd-addressed 8520 CIA chip. These lines are CNT, for the keyboard clock (input from keyboard), and SP, for keyboard data (input or output).

HOW THE KEYBOARD DATA IS RECEIVED

The CNT line is used as a clock for the keyboard. On each transition of this line, one bit of data is clocked in from the keyboard. The keyboard sends this clock when each data bit that is to be sent is stable on the SP line. The clock is an active low pulse. The rising edge of this pulse clocks in the data.

The 8520 is set up to use the CNT line as a clock and the SP line as a data input to an internal serial shift register. Appendix F contains most of the data sheet for the 8520 and provides more information for interested parties.

After a data byte has been received from the keyboard, an interrupt (from the 8520) is issued to the processor. The keyboard waits for a handshake signal from the system before transmitting any more keystrokes. (The handshake is issued by the processor pulsing the SP line low for a minimum of 75 microseconds.)

If another keystroke is received before the previous one has been accepted by the processor, the keyboard-processor (internal to keyboard) holds a type-ahead buffer approximately 10 "keycodes" long. (Keycodes are explained in the next section).

TYPE OF DATA RECEIVED

The keyboard data is *not* received in the form of ASCII characters. Instead, for maximum versatility, it is received in the form of keycodes. These codes include not only the down-transition of the key, but also the up-transition. This allows your software to use both sets of information to determine exactly what is happening on the keyboard.

Here is a list of the hexadecimal values that are assigned to the keyboard. A downstroke of the key transmits the value shown here. An upstroke of the key transmits this value plus $80. The picture of the keyboard at the end of this section shows the positions that correspond to the description in the paragraphs below.

The 128 possible key codes are arranged into the logical groups shown below.

00-3F hex

These are key codes assigned to specific positions on the main body of the keyboard and the numeric pad that contain graphic keys (that is, "A", but not "Tab"). The key positions would generally be labeled with country-dependent keys. These keycodes are best described positionally as shown in figure 8-3 at the end of the keyboard section.

40-4F hex

These are key codes with specific meanings common to most keyboards:

40	Space
41	Backspace
42	Tab
43	Enter (numeric pad)
44	Return
45	Escape
46	Delete
4A	Numeric pad
4C	Cursor up
4D	Cursor down
4E	Cursor forward
4F	Cursor backward

50-5F hex

Key codes for function keys:

50-59	Function keys F1-F10
5F	Help

60-67 hex

Key codes for qualifier keys:

60	Left shift
61	Right shift
62	Caps lock
63	Control
64	Left ALT
65	Right ALT
66	Left Amiga (command)
67	Right Amiga (command)

68-77 hex

Unassigned.

F0-FF hex

These key codes are used for 6500/01-68000 communication, and are not associated with a keystroke. They have no key transition flag, and are therefore described completely by 8-bit codes:

F9	Last key code bad, next key is same code retransmitted
FA	Keyboard key buffer overflow
FC	Keyboard self-test fail
FD	Initiate power-up key stream (for stuck keys)
FE	Terminate key stream (from FD)

These key codes may be filtered out by the drivers.

LIMITATIONS OF THE KEYBOARD

The Amiga keyboard (see figure 8-3) is a matrix of rows and columns, with a key switch at each intersection. Because of this, it is subject to a phenomenon called "ghosting."

Ghosting means that certain combinations of keys pressed simultaneously will cause extra ("ghost") key codes to be transmitted. For example, press "A" and "S" simultaneously and hold them down. Notice that "A" and "S" are transmitted. While still holding them down, press "Z" and observe that both "X" and "Z" are transmitted. In this case, "X" is a ghost key.

The keyboard is designed so that this will never happen during normal typing, only when unusual key combinations like the one just described are pressed. Normally, the keyboard will appear to have "N-key rollover," which means that you will run out of fingers before generating a ghost character.

NOTE

> There are seven keys that are not part of the matrix, and thus do not contribute to generating ghosts. These keys are: CTRL (control), the two SHIFT keys, the two Amiga keys, and the two ALT keys.

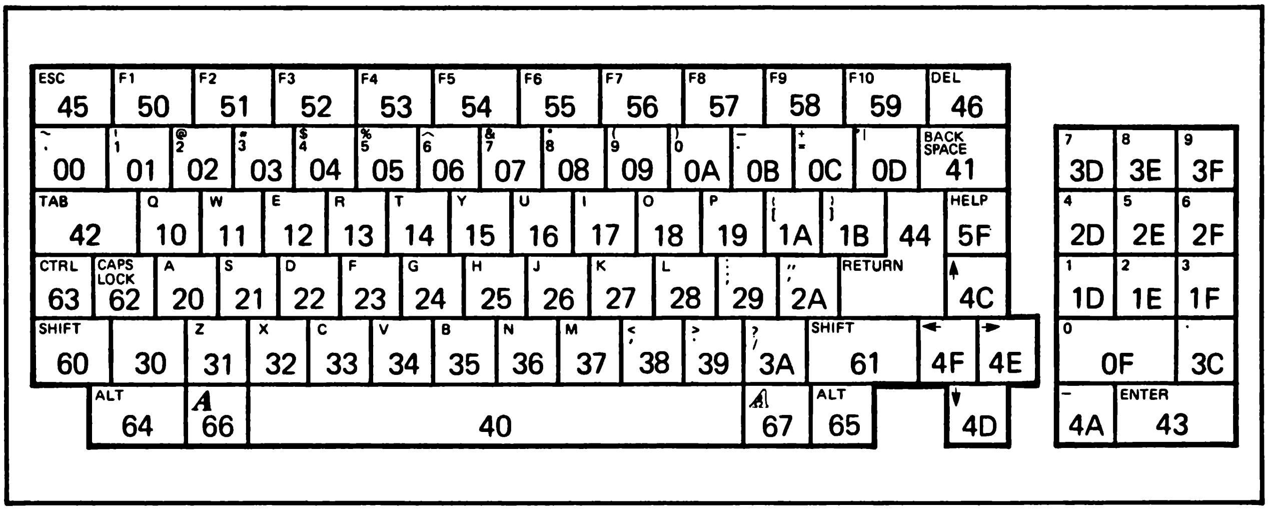

ESC 45
F1 50
F2 51
F3 52
F4 53
F5 54
F6 55
F7 56
F8 57
F9 58
F10 59
DEL 46
~ ` 00
! 1 01
@ 2 02
# 3 03
$ 4 04
% 5 05
^ 6 06
& 7 07
* 8 08
(9 09
) 0 0A
_ - 0B
+ = 0C
| \ 0D
BACK SPACE 41
TAB 42
Q 10
W 11
E 12
R 13
T 14
Y 15
U 16
I 17
O 18
P 19
{ [1A
}] 1B
44
HELP 5F
CTRL 63
CAPS LOCK 62
A 20
S 21
D 22
F 23
G 24
H 25
J 26
K 27
L 28
: ; 29
" ' 2A
RETURN
↑ 4C
SHIFT 60
30
Z 31
X 32
C 33
V 34
B 35
N 36
M 37
< , 38
> . 39
? / 3A
SHIFT 61
← 4F
→ 4E
ALT 64
A 66
40
A 67
ALT 65
↓ 4D
7 3D
8 3E
9 3F
4 2D
5 2E
6 2F
1 1D
2 1E
3 1F
0 0F
. 3C
- 4A
ENTER 43

Figure 8-3: The Amiga Keyboard, Showing Keycodes in Hexadecimal

Parallel Input/Output Interface

The general-purpose parallel interface is a 25-pin male connector on the back panel of the computer. This connector is generally used for a parallel printer.

For pin connections, see appendix E.

Serial Interface

A 25-pin D-type female connector on the back panel of the computer serves as the general purpose serial interface. This connector can drive a wide range of different peripherals, including an external modem or a serial printer.

For pin connections, see appendix E.

INTRODUCTION TO SERIAL CIRCUITRY

The circuit that controls the serial link to the outside world is called a UART, which is short for Universal Asynchronous Receiver/Transmitter. The UART is able to communicate at baud rates (bit-rate of transmission of data) that you preset. It can receive or send data with a programmable length of eight or nine bits.

The UART is also capable of detecting overrun errors, which occur when some other system sends in data faster than you remove it from the data-receive register. There are also status bits that you can read to find out when the receive buffer is full or when the transmit buffer is empty. An additional status bit is provided that indicates "all bits sent." All of these topics are discussed below.

SETTING THE BAUD RATE

Baud rate (rate of transmission) is controlled by the contents of the register named SERPER. Bits 14-0 of SERPER are the baud-rate divider bits. If you consider the contents of these bits to be the number N, then N+1 color clocks (each 279.4 ns) occur between samples of the state of the input pin (for receive) or between transmissions of output bits (in the transmit mode).

SETTING THE RECEIVE MODE

The number of bits that are to be received before the system tells you that the receive register is full may be defined either as eight or nine. In either case, the receive circuitry expects to see one start bit, eight or nine data bits, and at least one stop bit.

Receive mode is set by bit 15 of SERPER. Bit 15 is a 1 if if you chose nine data bits for the receive-register full signal, and a 0 if you chose eight data bits. The normal state of this bit for most receive applications is a 0.

SERPER is a write-only register.

CONTENTS OF THE RECEIVE DATA REGISTER

The serial input data-receive register is 16 bits wide. It contains not only the input data received but also certain status bits, which are explained below.

The data bit positions defined for read-data are taken from the "back-up" register, which is connected to the receive-data serial shift register.

The data is received, one bit at a time, into a serial-to-parallel shift register. When the proper number of bits has been received, the contents of this register are transferred to the serial data read register (SERDATR) shown in table 8-10, and you are signaled that there is data ready for you.

The back-up register is called that because immediately after the transfer of data takes place, the receive shift register again becomes ready to accept new data. After receiving the receiver-full interrupt, therefore, you will have up to one full character-receive time (8 to 10 bit times) to accept the data and clear the interrupt.

Table 8-10 shows the definitions of the various bit positions within SERDATR.

Table 8-10: SERDATR Register

Bit Number	**Name**	**Function**
15	OVRUN	OVERRUN bit (Mirror—also appears in the interrupt request register.) Indicates that another byte of data was received before the previous byte was picked up by the processor. To prevent this condition, it is necessary to reset the RBF bit (bit 11) (receive-buffer-full) in the interrupt request register (INTREQ).
14	RBF	READ BUFFER FULL (Mirror—also appears in the interrupt request register.) When it is a 1, it says that there is data ready to be picked up by the processor. After reading the contents of this data register, you must reset the RBF bit in INTREQ to prevent an overrun.
13	TBE	TRANSMIT BUFFER EMPTY (Not a mirror—interrupt occurs when the buffer *becomes* empty.) When it is a 1, the data in the output data register (SERDAT) has been transferred to the serial output shift register, so SERDAT is ready to accept another output word. This is also true when the buffer *is* empty. This bit is normally used for full-duplex operation.
12	TSRE	TRANSMIT SHIFT REGISTER EMPTY When this bit is a 1, the output shift register has completed its task, all data has been transmitted, and the register is now idle. If you stop writing data into the output register (SERDAT), then this bit will become a 1 after both the word currently in the shift register *and* the word placed into SERDAT have been transmitted. This bit is normally used for half-duplex operation.

11	RXD	Direct read of RXD pin on Paula chip.
10		Not used at this time
9	STP	Stop bit if 9 data bits are specified for receive.
8	STP DB8	Stop bit if 8 data bits are specified for receive, OR 9th data bit if 9 bits are specified for receive.
7-0	DB7-DB0	Low 8 data bits of received data. Data is TRUE (data you read is the same polarity as the data expected).

HOW OUTPUT DATA IS TRANSMITTED

You send data out on the transmit lines by writing into the serial data output register (SERDAT). This register is write-only.

Data will be sent out at the same rate as you have established for the read, and this data is contained in the serial data period register (SERPER) shown above. Immediately after you write the data into this register, the system will begin the transmission at the baud rate you selected.

At the start of the operation, this data is transferred from SERDAT into a serial shift register. When the transfer to the serial shift register has been completed, SERDAT can accept new data; the TBE interrupt signals this fact.

Data will be moved out of the shift register, one bit during each time interval, starting with the least significant bit. The shifting continues until, following the last shift, the UART detects the condition "shift-register-empty," which means that only 0s remain in the register.

SERDAT is a 16-bit register that allows you to control the format (appearance) of the transmitted data. To form a typical data sequence, such as one start bit, eight data bits, and two stop bits, you write into SERDAT the contents shown in figures 8-4 and 8-5.

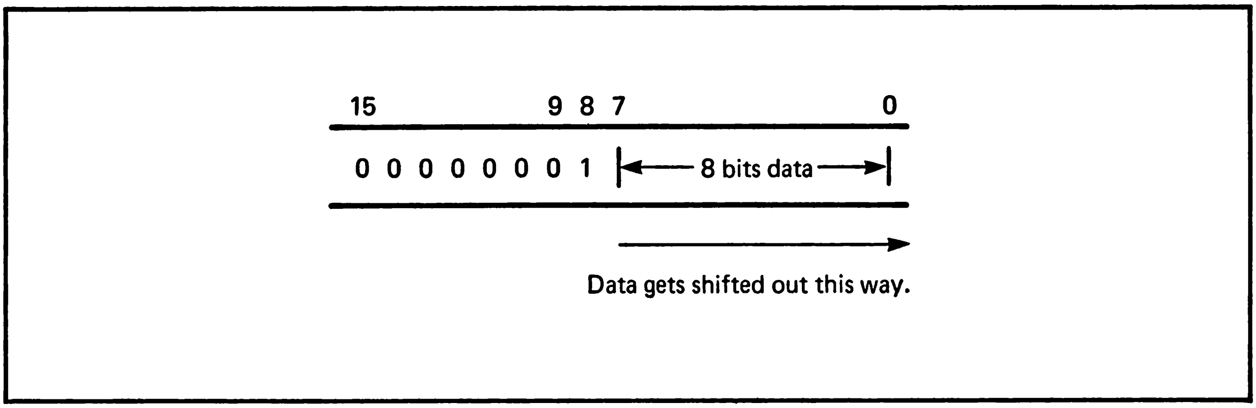

15
9 8 7
0
0 0 0 0 0 0 0 1
8 bits data
Data gets shifted out this way.

Figure 8-4: Starting Appearance of SERDAT and Shift Register

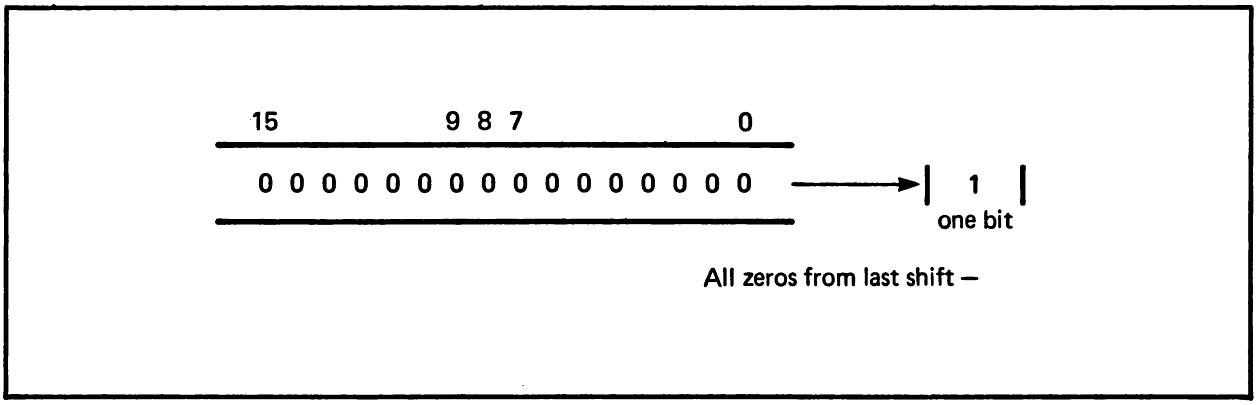

15
9 8 7
0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1
one bit
All zeros from last shift –

Figure 8-5: Ending Appearance of Shift Register

The register stops shifting and signals "shift register empty" (TSRE) when there is a 1 bit present in the bit-shifted-out position *and* the rest of the contents of the shift register are 0s. When new nonzero contents are transferred into this register, shifting begins again.

SPECIFYING THE REGISTER CONTENTS

You should write the data you wish to transmit as the low 8 (or 9 if you wish) bits of this output register (SERDAT). Above the data bits (in bits 8 and above or bits 9 and above) you write 1 bits for however many stop bits you transmit with the data.

Normally, you send either one or two stop bits. (See figure 8-4.)

The transmission of the start bit is independent of the contents of this register. One start bit is automatically generated before the first bit (bit 0) of the data is sent.

Writing this register starts the data transmission. If this register is written with all zeros, *no* data transmission is initiated.

Audio Output Connections

The Amiga has two different forms of audio output for the audio channels:

- Stereo output jacks

  A pair of "RCA" jacks, designed to be connected to a stereo amplifier.

- RF-Audio

  The channel 3/4 RF modulator will provide sound through the speaker of your television set when the television is used to provide the computer's display. Both channels of audio are provided at this connector. However, the RF modulator on initial shipments of Amiga computers combines the signals and transmits monaural sound.

Display Output Connections

A 23-pin connector on the back of the Amiga contains signals for two different types of video output. A separate cable assembly will be made up for each different type of video. The types are listed below.

- RGB Monitors ("analog RGB"). Provides four outputs, specifically red (R), green (G), blue (B), and sync. They can generate up to 4,096 different colors on-screen simultaneously using the circuitry presently available on the Amiga.

- Digital RGB Monitors. Provides four outputs, distinct from those shown above, named red (R), green (G), blue (B), half-intensity (I), and sync. All output levels are logic levels (0 or 1). These outputs allow up to 15 possible color combinations, where the values 0000 and 0001 map to the same output value. (Half intensity with no color present is the same as full intensity, no color.)

Appendix A

Register Summary—Alphabetical Order

This appendix contains a short summary, in alphabetical order, of the register set and the usages of the individual bits.

The addresses shown here are used by the special chips (called "Agnus", "Denise", and "Paula") for transferring data among themselves. Also, the Copper uses these addresses for writing to the special chip registers. To write to these registers with the 68000, calculate the 68000 address using this formula:

68000 address = (chip address) + $DFF000

For example, for the 68000 to write to ADKCON (address = $09E), the address would be $DFF09E.

```
                            Agnus/
                    Read/   Denise/
Register Address    Write   Paula         Function
-------- -------    ------  -------       --------

ADKCON    09E        W       P        Audio, disk, control write
ADKCONR   010        R       P        Audio, disk, control read

                  BIT#    USE
                  ----    -----------------------------------
                  15     SET/CLR      Set/clear control bit. Determines if bits
                                      written with a 1 get set or cleared. Bits
                                      written with a zero are always unchanged.
                  14-13  PRECOMP 1-0    CODE  PRECOMP VALUE
                                        ----  -------------
                                         00    none
                                         01    140 ns
                                         10    280 ns
                                         11    560 ns
                  12 MFMPREC          ( 1=MFM precomp 0=GCR precomp)
                  11 UARTBRK          Forces a UART break (clears TXD) if true.
                  10 WORDSYNC         Enables disk read synchronizing on a word
                                      equal to DISK SYNC CODE, located in
                                      address (3F)*2.
                  09 MSBSYNC          Enables disk read synchronizing on the MSB
                                      (most signif bit). Appl type GCR.
                  08 FAST     Disk data clock rate control 1=fast(2us) 0=slow(4us).
                               (fast for MFM, slow for MFM or GCR)

                  07 USE3PN Use audio channel 3 to modulate nothing.
                  06 USE2P3 Use audio channel 2 to modulate period of channel 3.
                  05 USE1P2 Use audio channel 1 to modulate period of channel 2.
                  04 USE0P1 Use audio channel 0 to modulate period of channel 1.

                  03 USE3VN Use audio channel 3 to modulate nothing.
                  02 USE2V3 Use audio channel 2 to modulate volume of channel 3.
                  01 USE1V2 Use audio channel 1 to modulate volume of channel 2.
                  00 USE0V1 Use audio channel 0 to modulate volume of channel 1.
                     NOTE:  If both period and volume are modulated on the
                     same channel, the period and volume will be alternated.
                     First word xxxxxxxx V6-V0 , Second word P15-P0 (etc)

AUDxLCH   0A0  W   A      Audio channel x location (high 3 bits)
AUDxLCL   0A2  W   A      Audio channel x location (low 15 bits)
               This pair of registers contains the 18 bit starting address
               (location) of audio channel x (x=0,1,2,3) DMA data.
               This is not a pointer register and therefore needs
               to be reloaded only if a different memory location is to
               be outputted.

AUDxLEN   0A4  W       P  Audio channel x length
               This register contains the length (number of words) of
               audio channel x DMA data.
```

```
AUDxPER    0A6  W      P  Audio channel x Period
                This register contains the period (rate) of
                audio channel x DMA data transfer.
                The minimum period is 124 color clocks. This means
                that the smallest number that should be placed in
                this register is 124 decimal.  This corresponds to
                a maximum sample frequency of 28.86 khz.

AUDxVOL    0A8  W      P  Audio channel x volume
                This register contains the volume setting for
                audio channel x.  Bits 6,5,4,3,2,1,0 specify 65
                linear volume levels as shown below.

                Bit#   Use
                ----   -------------------------
                 15-07 Not used
                 06    Forces volume to max (64 ones, no zeros)
                 05-00 Sets one of 64 levels (000000=no output
                       (111111=63 1s, one 0)

AUDxDAT    0AA  W      P  Audio channel x data
                This register is the audio channel x (x=0,1,2,3)
                DMA data buffer.  It contains 2 bytes of data that
                are each 2's complement and are outputted
                sequentially (with digital-to-analog conversion)
                to the audio output pins.  (LSB = 3 MV)  The DMA
                controller automatically transfers data to this
                register from RAM.  The processor can also write
                directly to this register.  When the DMA data is
                finished (words outputted=length) and the data in
                this register has been used, an audio channel
                interrupt request is set.

BLTxPTH    050  W  A      Blitter pointer to x (high 3 bits)
BLTxPTL    052  W  A      Blitter pointer to x (low 15 bits)
                This pair of registers contains the 18-bit address
                of blitter source (x=A,B,C) or destination (x=D)
                DMA data.  This pointer must be preloaded with the
                starting address of the data to be processed by
                the blitter. After the blitter is finished, it
                will contain the last data address (plus increment
                and modulo).
                LINE DRAW   BLTAPTL is used as an accumulator
                LINE DRAW   register and must be preloaded with
                LINE DRAW   the starting value of (2Y-X) where
                LINE DRAW   Y/X is the line slope.  BLTCPT and
                LINE DRAW   BLTDPT (both H and L) must be
                LINE DRAW   preloaded with the starting address
                LINE DRAW   of the line.

BLTxMOD    064  W  A      Blitter modulo x
                This register contains the modulo for blitter
                source (x=A,B,C) or destination (x=D).  A modulo
                is a number that is automatically added to the
                address at the end of each line, to make the
```

```
                    address point to the start of the next line.  Each
                    source or destination has its own modulo, allowing
                    each to be a different size, while an identical
                    area of each is used in the blitter operation.
                    LINE DRAW   BLTAMOD and BLTBMOD are used as slope
                    LINE DRAW   storage registers and must be preloaded
                    LINE DRAW   with the values (4Y-4X) and (4Y)
                    LINE DRAW   respectively.  Y/X= line slope.
                    LINE DRAW   BLTCMOD and BLTDMOD must both be
                    LINE DRAW   preloaded with the width (in bytes)
                    LINE DRAW   of the image into which the line is
                    LINE DRAW   being drawn (normally two times the
                    LINE DRAW   screen width in words).

BLTAFWM     044 W  A            Blitter first-word mask for source A
BLTALWM     046 W  A            Blitter last-word mask for source A
                    The patterns in these two registers are ANDed with
                    the first and last words of each line of data from
                    source A into the blitter. A zero in any bit
                    overrides data from source A.  These registers
                    should be set to all 1s for fill mode or for
                    line-drawing mode.

BLTxDAT     074 W  A            Blitter source x data register
                    This register holds source x (x=A,B,C) data for
                    use by the blitter. It is normally loaded by the
                    blitter DMA channel; however, it may also be
                    preloaded by the microprocessor.
                    LINE DRAW   BLTADAT is used as an index register
                    LINE DRAW   and must be preloaded with 8000.
                    LINE DRAW   BLTBDAT is used for texture; it must
                    LINE DRAW   be preloaded with FF if no texture
                    LINE DRAW    (solid line) is desired.

BLTDDAT                          Blitter destination data register
                    This register holds the data resulting from each
                    word of blitter operation until it is sent to a
                    RAM destination.  This is a dummy address and
                    cannot be read by the micro.  The transfer is
                    automatic during blitter operation.

BLTCON0    040  W  A       Blitter control register 0
BLTCON1    042  W  A       Blitter control register 1
                    These two control registers are used together to
                    control blitter operations.  There are two basic
                    modes, area and line, which are selected by bit
                    0 of BLTCON1, as shown below.
```

AREA MODE ("normal")

BIT#	BLTCON0	BLTCON1
----	-------	-------
15	ASH3	BSH3
14	ASH2	BSH2
13	ASH1	BSH1
12	ASA0	BSH0
11	USEA	X
10	USEB	X
09	USEC	X
08	USED	X
07	LF7	X
06	LF6	X
05	LF5	X
04	LF4	EFE
03	LF3	IFE
02	LF2	FCI
01	LF1	DESC
00	LF0	LINE(=0)

ASH3-0	Shift value of A source
BSH3-0	Shift value of B source
USEA	Mode control bit to use source A
USEB	Mode control bit to use source B
USEC	Mode control bit to use source C
USED	Mode control bit to use destination D
LF7-0	Logic function minterm select lines
EFE	Exclusive fill enable
IFE	Inclusive fill enable
FCI	Fill carry input
DESC	Descending (decreasing address) control bit
LINE	Line mode control bit (set to 0)

LINE MODE (line draw)

	BIT#	BLTCON0	BLTCON1
LINE DRAW			
LINE DRAW			
LINE DRAW	BIT#	BLTCON0	BLTCON1
LINE DRAW	----	-------	-------
LINE DRAW	15	START3	TEXTURE3
LINE DRAW	14	START2	TEXTURE2
LINE DRAW	13	START1	TEXTURE1
LINE DRAW	12	START0	TEXTURE0
LINE DRAW	11	1	0
LINE DRAW	10	0	0
LINE DRAW	09	1	0
LINE DRAW	08	1	0
LINE DRAW	07	LF7	0
LINE DRAW	06	LF6	SIGN
LINE DRAW	05	LF5	0 (Reserved)
LINE DRAW	04	LF4	SUD
LINE DRAW	03	LF3	SUL
LINE DRAW	02	LF2	AUL
LINE DRAW	01	LF1	SING
LINE DRAW	00	LF0	LINE(=1)
LINE DRAW			

```
            LINE DRAW   START3-0  Starting point of line
            LINE DRAW             (0 thru 15 hex)
            LINE DRAW   LF7-0     Logic function minterm
            LINE DRAW   select lines should be preloaded
            LINE DRAW   with 4A to select the equation
            LINE DRAW   D=(AC+ABC).  Since A contains a
            LINE DRAW   single bit true (8000), most bits
            LINE DRAW   will pass the C field unchanged
            LINE DRAW   (not A and C), but one bit will
            LINE DRAW   invert the C field and combine it
            LINE DRAW   with texture (A and B and not C).
            LINE DRAW   The A bit is automatically moved
            LINE DRAW   across the word by the hardware.
            LINE DRAW
            LINE DRAW   LINE   Line mode control bit
            LINE DRAW          (set to 1)
            LINE DRAW   SIGN   Sign flag
            LINE DRAW   0      Reserved for new mode
            LINE DRAW   SING   Single bit per horizontal
            LINE DRAW          line for use with subsequent
            LINE DRAW          area fill
            LINE DRAW   SUD    Sometimes up or down (=AUD*)
            LINE DRAW   SUL    Sometimes up or left
            LINE DRAW   AUL    Always up or left
            LINE DRAW   The 3 bits above select the octant
            LINE DRAW   for line drawing:
            LINE DRAW          OCT    SUD  SUL  AUL
            LINE DRAW          ---    ---  ---  ---
            LINE DRAW           0      1    1    0
            LINE DRAW           1      0    0    1
            LINE DRAW           2      0    1    1
            LINE DRAW           3      1    1    1
            LINE DRAW           4      1    0    1
            LINE DRAW           5      0    1    0
            LINE DRAW           6      0    0    0
            LINE DRAW           7      1    0    0
            LINE DRAW   The "B" source is used for
            LINE DRAW   texturing the drawn lines.
BLTSIZE   058 W  A       Blitter start and size (window width,
                         height)
            This register contains the width and height of
            the blitter operation (in line mode, width must
            = 2, height = line length).  Writing to this
            register will start the blitter, and should be
            done last, after all pointers and control
            registers have been initialized.
            BIT#  15,14,13,12,11,10,09,08,07,06,05,04,03,02,01,00
                  h9 h8 h7 h6 h5 h4 h3 h2 h1 h0,w5 w4 w3 w2 w1 w0
            h=height=vertical lines (10 bits=1024 lines max)
            w=width =horizontal pixels (6 bits=64 words=1024 pixels max)
            LINE DRAW  BLTSIZE controls the line length and starts
            LINE DRAW  the line draw when written to.  The h field
            LINE DRAW  controls the line length (10 bits gives
            LINE DRAW  lines up to 1024 dots long).  The w field
            LINE DRAW  must be set to 02 for all line drawing.
```

```
BPLxPTH   0E0  W  A     Bit plane x pointer (high 3 bits)
BPLxPTL   0E2  W  A     Bit plane x pointer (low 15 bits)
               This pair of registers contains the 18-bit pointer to
               the address of bit-plane x (x=1,2,3,4,5,6) DMA data.
               This pointer must be reinitialized by the processor
               or copper to point to the beginning of bit plane data
               every vertical blank time.

BPLxDAT   110  W    D   Bit plane x data (parallel-to-serial
                            convert)
               These registers receive the DMA data fetched from
               RAM by the bit plane address pointers described
               above.  They may also be written by either
               microprocessor.  They act as a six-word parallel-
               to-serial buffer for up to six memory bit planes
               (x=1-6).  The parallel-to-serial conversion is
               triggered whenever bit plane #1 is written,
               indicating the completion of all bit planes for
               that word (16 pixels).  The MSB is output first,
               and is, therefore, always on the left.

BPL1MOD   108  W  A     Bit plane modulo (odd planes)
BPL2MOD   10A  W  A     Bit Plane modulo (even planes)
               These registers contain the modulos for the odd
               and even bit planes.  A modulo is a number that is
               automatically added to the address at the end of
               each line, so that the address then points to the
               start of the next line.
               Since they have separate modulos, the odd and even
               bit planes may have sizes that are different from
               each other, as well as different from the display
               window size.


BPLCON0   100  W  A D   Bit plane control register (misc.
                             control bits)
BPLCON1   102  W    D   Bit plane control register
                             (horizontal scroll control)
BPLCON2   104  W    D   Bit Plane control register
                             (video priority control)
               These registers control the operation of the
               bit planes and various aspects of the display.
               BIT#     BPLCON0    BPLCON1    BPLCON2
               ----     --------   --------   --------
               15       HIRES       X          X
               14       BPU2        X          X
               13       BPU1        X          X
               12       BPU0        X          X
               11       HOMOD       X          X
               10       DBLPF       X          X
               09       COLOR       X          X
               08       GAUD        X          X
               07        X         PF2H3       X
               06        X         PF2H2      PF2PRI
               05        X         PF2H1      PF2P2
               04        X         PF2H0      PF2P1
               03       LPEN       PF1H3      PF2P0
```

02	LACE	PF1H2	PF1P2
01	ERSY	PF1H1	PF1P1
00	X	PF1H0	PF1P0

HIRES=High-resolution (640) mode
BPU =Bit plane use code 000-110 (NONE through 6 inclusive)
HOMOD=Hold-and-modify mode
DBLPF=Double playfield (PF1=odd PF2=even bit planes)
COLOR=Composite video COLOR enable
GAUD=Genlock audio enable (muxed on BKGND pin during vertical blanking
LPEN =Light pen enable (reset on power up)
LACE =Interlace enable (reset on power up)
ERSY =External resync (HSYNC, VSYNC pads become inputs) (reset on power up)
PF2PRI=Playfield 2 (even planes) has priority over (appears in front of) playfield 1 (odd planes).
PF2P=Playfield 2 priority code (with respect to sprites)
PF1P=Playfield 1 priority code (with respect to sprites)
PF2H=Playfield 2 horizontal scroll code
PF1H=Playfield 1 horizontal scroll code

CLXCON 098 W D Collision control

This register controls which bit-planes are included (enabled) in collision detection and their required state if included. It also controls the individual inclusion of odd-numbered sprites in the collision detection by logically OR-ing them with their corresponding even-numbered sprite.

BIT#	FUNCTION	DESCRIPTION
15	ENSP7	Enable sprite 7 (ORed with sprite 6)
14	ENSP5	Enable sprite 5 (ORed with sprite 4)
13	ENSP3	Enable sprite 3 (ORed with sprite 2)
12	ENSP1	Enable sprite 1 (ORed with sprite 0)
11	ENBP6	Enable bit plane 6 (match required for collision)
10	ENBP5	Enable bit plane 5 (match required for collision)
09	ENBP4	Enable bit plane 4 (match required for collision)
08	ENBP3	Enable bit plane 3 (match required for collision)
07	ENBP2	Enable bit plane 2 (match required for collision)
06	ENBP1	Enable bit plane 1 (match required for collision)
05	MVBP6	Match value for bit plane 6 collision
04	MVBP5	Match value for bit plane 5 collision
03	MVBP4	Match value for bit plane 4 collision
02	MVBP3	Match value for bit plane 3 collision
01	MVBP2	Match value for bit plane 2 collision
00	MVBP1	Match value for bit plane 1 collision

NOTE: Disabled bit planes cannot prevent collisions. Therefore if all bit planes are disabled, collisions will be continuous, regardless of the match values.

CLXDAT 00E R D Collision data register (read and clear)

This address reads (and clears) the collision detection register. The bit assignments are below.

NOTE: Playfield 1 is all odd-numbered enabled bit planes. Playfield 2 is all even-numbered enabled bit planes

BIT#	COLLISIONS REGISTERED
15	not used
14	Sprite 4 (or 5) to sprite 6 (or 7)
13	Sprite 2 (or 3) to sprite 6 (or 7)
12	Sprite 2 (or 3) to sprite 4 (or 5)
11	Sprite 0 (or 1) to sprite 6 (or 7)
10	Sprite 0 (or 1) to sprite 4 (or 5)
09	Sprite 0 (or 1) to sprite 2 (or 3)
08	Playfield 2 to sprite 6 (or 7)
07	Playfield 2 to sprite 4 (or 5)
06	Playfield 2 to sprite 2 (or 3)
05	Playfield 2 to sprite 0 (or 1)
04	Playfield 1 to sprite 6 (or 7)
03	Playfield 1 to sprite 4 (or 5)
02	Playfield 1 to sprite 2 (or 3)
01	Playfield 1 to sprite 0 (or 1)
00	Playfield 1 to playfield 2

COLORxx 180 W D Color table xx

There are 32 of these registers (xx=00-31) and they are sometimes collectively called the "color palette." They contain 12-bit codes representing red, green, and blue colors for RGB systems. One of these registers at a time is selected (by the BPLxDAT serialized video code) for presentation at the RGB video output pins. The table below shows the color register bit usage.

BIT#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
RGB	X	X	X	X	R3	R2	R1	R0	G3	G2	G1	G0	B3	B2	B1	B0

B=blue, G=green, R=red,

COPCON 02E W A Copper control register

This is a 1-bit register that when set true, allows the Copper to access the blitter hardware. This bit is cleared by power-on reset, so that the Copper cannot access the blitter hardware.

BIT#	NAME	FUNCTION
01	CDANG	Copper danger mode. Allows Copper access to blitter if true.

COPJMP1 088 S A Copper restart at first location
COPJMP2 08A S A Copper restart at second location

These addresses are strobe addresses. When written to, they cause the Copper to jump indirect using the address contained in the first or second location registers described below. The Copper itself can write to these addresses, causing its own jump indirect.

COP1LCH 080 W A Copper first location register (high 3 bits)
COP1LCL 082 W A Copper first location register (low 15 bits)
COP2LCH 084 W A Copper second location register (high 3 bits)
COP2LCL 086 W A Copper second location register (low 15 bits)

These registers contain the jump addresses described above.

COPINS 08C W A Copper instruction fetch identify

This is a dummy address that is generated by the Copper whenever it is loading instructions into its own instruction register. This actually occurs every Copper cycle except for the second (IR2) cycle of the MOVE instruction. The three types of instructions are shown below.

MOVE Move immediate to destination.
WAIT Wait until beam counter is equal to, or greater than. (keeps Copper off of bus until beam position has been reached).
SKIP Skip if beam counter is equal to or greater than (skips following MOVE instruction unless beam position has been reached).

	MOVE		WAIT UNTIL		SKIP IF	
BIT#	IR1	IR2	IR1	IR2	IR1	IR2
15	X	RD15	VP7	BFD *	VP7	BFD *
14	X	RD14	VP6	VE6	VP6	VE6
13	X	RD13	VP5	VE5	VP5	VE5
12	X	RD12	VP4	VE4	VP4	VE4
11	X	RD11	VP3	VE3	VP3	VE3
10	X	RD10	VP2	VE2	VP2	VE2
09	X	RD09	VP1	VE1	VP1	VE1
08	DA8	RD08	VP0	VE0	VP0	VE0
07	DA7	RD07	HP8	HE8	HP8	HE8
06	DA6	RD06	HP7	HE7	HP7	HE7
05	DA5	RD05	HP6	HE6	HP6	HE6
04	DA4	RD04	HP5	HE5	HP5	HE5
03	DA3	RD03	HP4	HE4	HP4	HE4
02	DA2	RD02	HP3	HE3	HP3	HE3
01	DA1	RD01	HP2	HE2	HP2	HE2
00	0	RD00	1	0	1	1

IR1=First instruction register
IR2=Second instruction register
DA =Destination address for MOVE instruction. Fetched during IR1 time, used during IR2 time on RGA bus.
RD =RAM data moved by MOVE instruction at IR2 time directly from RAM to the address given by the DA field.
VP =Vertical beam position comparison bit.
HP =Horizontal beam position comparison bit.
VE =Enable comparison (mask bit).
HE =Enable comparison (mask bit).
* NOTE BFD=Blitter finished disable. When this bit is true, the Blitter Finished flag will have no effect on the Copper. When this bit is zero, the Blitter Finished flag must be true (in addition to the rest of the bit comparisons) before the Copper can exit from its wait state or skip over an instruction. Note that the V7 comparison cannot be masked.

The Copper is basically a two-cycle machine that requests the bus only during odd memory cycles (4 memory cycles per instruction). This prevents collisions with display, audio, disk, refresh, and sprites, all of which use only even cycles. It therefore needs (and has) priority over only the blitter and microprocessor.

There are only three types of instructions: MOVE immediate, WAIT until, and SKIP if. All instructions (except for WAIT) require two bus cycles (and two instruction words). Since only the odd bus cycles are requested, four memory cycle times are required per instruction (memory cycles are 280 ns.)

There are two indirect jump registers, COP1LC and COP2LC. These are 18-bit pointer registers whose contents are used to modify the program counter for initialization or jumps. They are transferred to the program counter whenever strobe addresses COPJMP1 or COPJMP2 are written. In addition, COP1LC is automatically used at the beginning of each vertical blank time.

It is important that one of the jump registers be initialized and its jump strobe address hit after power-up but before Copper DMA is initialized. This insures a determined startup address and state.

DIWSTRT 08E W A Display window start (upper left vertical-horizontal position)
DIWSTOP 090 W A Display window stop (lower right vertical-horizontal position)

These registers control display window size and position by locating the upper left and lower right corners.

BIT# 15,14,13,12,11,10,09,08,07,06,05,04,03,02,01,00
USE V7 V6 V5 V4 V3 V2 V1 V0 H7 H6 H5 H4 H3 H2 H1 H0

DIWSTRT is vertically restricted to the upper 2/3 of the display (V8=0) and horizontally restricted to the left 3/4 of the display (H8=0).

DIWSTOP is vertically restricted to the lower 1/2 of the display (V8=/=V7) and horizontally restricted to the right 1/4 of the display (H8=1).

DDFSTRT 092 W A Display data fetch start (horiz. position)
DDFSTOP 094 W A Display data fetch stop (horiz. position)

These registers control the horizontal timing of the beginning and end of the bit plane DMA display data fetch. The vertical bit plane DMA timing is identical to the display windows described above.

The bit plane modulos are dependent on the bit plane horizontal size and on this data-fetch window size.

Register bit assignment

BIT# 15,14,13,12,11,10,09,08,07,06,05,04,03,02,01,00
USE X X X X X X X X H8 H7 H6 H5 H4 H3 X X

(X bits should always be driven with 0 to maintain upward compatibility)

The tables below show the start and stop timing for different register contents.

DDFSTRT (left edge of display data fetch)

PURPOSE	H8,H7,H6,H5,H4
Extra wide (max) *	0 0 1 0 1
Wide	0 0 1 1 0
Normal	0 0 1 1 1
Narrow	0 1 0 0 0

DDFSTOP (right edge of display data fetch)

PURPOSE	H8,H7,H6,H5,H4
Narrow	1 1 0 0 1
Normal	1 1 0 1 0
Wide (max)	1 1 0 1 1

DMACON 096 W A D P DMA control write (clear or set)
DMACONR 002 R A P DMA control (and blitter status) read

This register controls all of the DMA channels and contains blitter DMA status bits.

BIT#	FUNCTION	DESCRIPTION
15	SET/CLR	Set/clear control bit. Determines if bits written with a 1 get set or cleared. Bits written with a zero are unchanged.
14	BBUSY	Blitter busy status bit (read only)
13	BZERO	Blitter logic zero status bit (read only).
12	X	
11	X	
10	BLTPRI	Blitter DMA priority (over CPU micro) (also called "blitter nasty") (disables /BLS pin, preventing micro from stealing any bus cycles while blitter DMA is running).
09	DMAEN	Enable all DMA below
08	BPLEN	Bit plane DMA enable
07	COPEN	Copper DMA enable
06	BLTEN	Blitter DMA enable
05	SPREN	Sprite DMA enable
04	DSKEN	Disk DMA enable
03	AUD3EN	Audio channel 3 DMA enable
02	AUD2EN	Audio channel 2 DMA enable
01	AUD1EN	Audio channel 1 DMA enable
00	AUD0EN	Audio channel 0 DMA enable

DSKPTH 020 W A Disk pointer (high 3 bits)
DSKPTL 022 W A Disk pointer (low 15 bits)

This pair of registers contains the 18-bit address of disk DMA data. These address registers must be initialized by the processor or Copper before disk DMA is enabled.

DSKLEN 024 W P Disk length

This register contains the length (number of words) of disk DMA data. It also contains two control bits, a DMA enable bit, and a DMA direction (read/write) bit.

BIT#	FUNCTION	DESCRIPTION
15	DMAEN	Disk DMA enable
14	WRITE	Disk write (RAM to disk) if 1
13-0	LENGTH	Length (# of words) of DMA data.

DSKDAT 026 W P Disk DMA data write
DSKDATR 008 ER P Disk DMA data read (early read dummy address)

This register is the disk DMA data buffer. It contains two bytes of data that are either sent (written) to or received (read) from the disk. The write mode is enabled by bit 14 of the LENGTH register. The DMA controller automatically transfers data to or from this register and RAM, and when the DMA data is finished (length=0) it causes a disk block interrupt. See interrupts

DSKBYTR 01A R P Disk data byte and status read

This register is the disk-microprocessor data buffer. Data from the disk (in read mode) is loaded into this register one byte at a time, and bit 15 (DSKBYT) is set true.

BIT#		
15	DSKBYT	Disk byte ready (reset on read)
14	DMAON	Mirror of bit 15 (DMAEN) in DSKLEN, ANDed with Bit09 (DMAEN) in DMACON
13	DISKWRITE	Mirror of bit 14 (WRITE) in DSKLEN
12	WORDEQUAL	This bit true only while the DSKSYNC register equals the data from disk.
11-08	X	Not used
07-00	DATA	Disk byte data

DSKSYNC 07E W P Disk sync register, holds the match code for disk read synchronization. See ADKCON bit 10.

INTREQ 09C W P Interrupt request bits (clear or set)
INTREQR 01E R P Interrupt request bits (read)

This register contains interrupt request bits (or flags). These bits may be polled by the processor; if enabled by the bits listed in the next register, they may cause processor interrupts. Both a set and clear operation are required to load arbitrary data into this register. These status bits are not automatically reset when the interrupt is serviced, and must be reset when desired by writing to this address. The bit assignments are identical to the enable register below.

INTENA 09A W P Interrupt enable bits (clear or set bits)
INTENAR 01C R P Interrupt enable bits (read)

This register contains interrupt enable bits. The bit assignment for both the request and enable registers is given below.

BIT#	FUNCT	LEVEL	DESCRIPTION
15	SET/CLR		Set/clear control bit. Determines if bits written with a 1 get set or cleared. Bits written with a zero are always unchanged.
14	INTEN		Master interrupt (enable only, no request)
13	EXTER	6	External interrupt
12	DSKSYN	5	Disk sync register (DSKSYNC) matches disk data
11	RBF	5	Serial port receive buffer full
10	AUD3	4	Audio channel 3 block finished
09	AUD2	4	Audio channel 2 block finished
08	AUD1	4	Audio channel 1 block finished
07	AUD0	4	Audio channel 0 block finished
06	BLIT	3	Blitter finished
05	VERTB	3	Start of vertical blank
04	COPER	3	Copper
03	PORTS	2	I/O ports and timers
02	SOFT	1	Reserved for software-initiated interrupt
01	DSKBLK	1	Disk block finished
00	TBE	1	Serial port transmit buffer empty

JOY0DAT 00A R D Joystick-mouse 0 data (left vertical, horizontal)
JOY1DAT 00C R D Joystick-mouse 1 data (right vertical, horizontal)

These addresses each read a pair of 8-bit mouse counters. 0=left controller pair, 1=right controller pair (four counters total). The bit usage for both left and right addresses is shown below. Each counter is clocked by signals from two controller pins. Bits 1 and 0 of each counter may be read to determine the state of these two clock pins. This allows these pins to double as joystick switch inputs.

Mouse counter usage:
(pins 1,3=Yclock, pins 2,4=Xclock)

BIT#	15,14,13,12,11,10,09,08	07,06,05,04,03,02,01,00
0DAT	Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0	X7 X6 X5 X4 X3 X2 X1 X0
1DAT	Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0	X7 X6 X5 X4 X3 X2 X1 X0

The following table shows the mouse/joystick connector pin usage. The pins (and their functions) are sampled (multiplexed) into the DENISE chip during the clock times shown in the table. This table is for reference only and should not be needed by the programmer. (Note that the joystick functions are all "active low" at the connector pins.)

Conn Pin	Joystick Function	Mouse Function	Sampled by DENISE Pin	Name	Clock
L1	FORW*	Y	38	M0V at	CCK
L3	LEFT*	YQ	38	M0V at	CCK*
L2	BACK*	X	9	M0H at	CCK
L4	RIGH*	XQ	9	M0H at	CCK*
R1	FORW*	Y	39	M1V at	CCK
R3	LEFT*	YQ	39	M1V at	CCK*
R2	BACK*	X	8	M1H at	CCK
R4	RIGH*	XQ	8	M1H at	CCK*

After being sampled, these connector pin signals are used in quadrature to clock the mouse counters. The LEFT and RIGHT joystick functions (active high) are directly available on the Y1 and X1 bits of each counter. In order to recreate the FORWARD and BACK joystick functions, however, it is necessary to logically combine (exclusive OR) the lower two bits of each counter. This is illustrated in the following table.

To detect	Read these counter bits
Forward	Y1 xor Y0 (BIT#09 xor BIT#08)
Left	Y1
Back	X1 xor X0 (BIT#01 xor BIT#00)
Right	X1

JOYTEST 036 W D Write to all four joystick-mouse counters at once.

Mouse counter write test data:

BIT#	15,14,13,12,11,10,09,08	07,06,05,04,03,02,01,00
0DAT	Y7 Y6 Y5 Y4 Y3 Y2 xx xx	X7 X6 X5 X4 X3 X2 xx xx
1DAT	Y7 Y6 Y5 Y4 Y3 Y2 xx xx	X7 X6 X5 X4 X3 X2 xx xx

POT0DAT 012 R P Pot counter data left pair (vert,horiz)
POT1DAT 014 R P Pot counter data right pair (vert,horiz)

These addresses each read a pair of 8-bit pot counters. (Four counters total.) The bit assignment for both addresses is shown below. The counters are stopped by signals from two controller connectors (left-right) with two pins each.

BIT#	15,14,13,12,11,10,09,08	07,06,05,04,03,02,01,00
RIGHT	Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0	X7 X6 X5 X4 X3 X2 X1 X0
LEFT	Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0	X7 X6 X5 X4 X3 X2 X1 X0

CONNECTORS				PAULA	
Loc.	Dir.	Sym	Pin	Pin#	Pin Name
RIGHT	Y	RY	9	36	(POT1Y)
RIGHT	X	RX	5	35	(POT1X)
LEFT	Y	LY	9	33	(POT0Y)
LEFT	X	LX	5	32	(POT0X)

POTGO 034 W P Pot port data write and start.

POTGOR 016 R P Pot port data read (formerly called POTINP). This register controls a 4-bit bi-directional I/O port that shares the same four pins as the four pot counters above.

BIT#	FUNCT	DESCRIPTION
15	OUTRY	Output enable for Paula pin 36
14	DATRY	I/O data Paula pin 36
13	OUTRX	Output enable for Paula pin 35
12	DATRX	I/O data Paula pin 35
11	OUTLY	Output enable for Paula pin 33
10	DATLY	I/O data Paula pin 33
09	OUTLX	Output enable for Paula pin 32
08	DATLX	I/O data Paula pin 32
07-01	0	Reserved for chip ID code (presently 0)
00	START	Start pots (dump capacitors, start counters)

REFPTR 028 W A Refresh pointer

This register is used as a dynamic RAM refresh address generator. It is writeable for test purposes only, and should never be written by the microprocessor.

SERDAT 030 W P Serial port data and stop bits write (transmit data buffer)

This address writes data to a transmit data buffer. Data from this buffer is moved into a serial shift register for output transmission whenever it is empty. This sets the interrupt request TBE (transmit buffer empty). A stop bit must be provided as part of the data word. The length of the data word is set by the position of the stop bit.

BIT#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
USE	0	0	0	0	0	0	S	D8	D7	D6	D5	D4	D3	D2	D1	D0

Note: S = stop bit = 1, D = data bits.

SERDATR 018 R P Serial port data and status read (receive data buffer)

This address reads data from a receive data buffer. Data in this buffer is loaded from a receiving shift register whenever it is full. Several interrupt request bits are also read at this address, along with the data, as shown below.

BIT#		
15	OVRUN	Serial port receiver overrun. Reset by resetting bit 11 of INTREQ.
14	RBF	Serial port receive buffer full (mirror).
13	TBE	Serial port transmit buffer empty (mirror).
12	TSRE	Serial port transmit shift register empty. Reset by loading into buffer.
11	RXD	RXD pin receives UART serial data for direct bit test by the microprocessor.
10	0	Not used
09	STP	Stop bit
08	STP-DB8	Stop bit if LONG, data bit if not.
07	DB7	Data bit
06	DB6	Data bit
05	DB5	Data bit
04	DB4	Data bit
03	DB3	Data bit
02	DB2	Data bit
01	DB1	Data bit
00	DB0	Data bit

```
SERPER     032 W     P  Serial port period and control
               This register contains the control bit LONG referred to
               above, and a 15-bit number defining the serial port
               baud rate. If this number is N, then the baud rate is
               1 bit every (N+1)*.2794  microseconds.
               BIT#
               15     LONG      Defines serial receive as 9-bit word.
               14-00  RATE      Defines baud rate=1/((N+1)*.2794 microsec.)

SPRxPTH    120 W  A     Sprite x pointer (high 3 bits)
SPRxPTL    122 W  A     Sprite x pointer (low 15 bits)
               This pair of registers contains the 18-bit address
               of sprite x (x=0,1,2,3,4,5,6,7) DMA data. These address
               registers must be initialized by the processor or Copper
               every vertical blank time.

SPRxPOS    140 W  A D   Sprite x vert-horiz start position data
SPRxCTL    142 W  A D   Sprite x vert stop position and control data
               These two registers work together as position, size and
               feature sprite-control registers.  They are usually loaded
               by the sprite DMA channel during horizontal blank;
               however, they may be loaded by either processor at any time.
               SPRxPOS register:
               BIT#  SYM   FUNCTION
               ----  ----  ----------------------------
               15-08 SV7-SV0 Start vertical value. High bit(SV8) is
                             in SPRxCTL register below.
               07-00 SH8-SH1 Start horizontal value. Low bit(SH0) is
                             in SPRxCTL register below.

               SPRxCTL register (writing this address disables sprite
                                horizontal comparator circuit):
               BIT#    SYM      FUNCTION
               ----    -------- ----------------------------
               15-08   EV7-EV0  End (stop) vertical value low 8 bits
               07      ATT      Sprite attach control bit (odd sprites)
               06-04    X       Not used
               02      SV8      Start vertical value high bit
               01      EV8      End (stop) vertical value high bit
               00      SH0      Start horizontal value low bit

SPRxDATA   144 W    D   Sprite x image data register A
SPRxDATB   146 W    D   Sprite x image data register B
               These registers buffer the sprite image data. They are
               usually loaded by the sprite DMA channel but may be
               loaded by either processor at any time.  When a
               horizontal comparison occurs, the buffers are dumped
               into shift registers and serially outputted to the
               display, MSB first on the left.
               NOTE: Writing to the A buffer enables (arms) the sprite.
               Writing to the SPRxCTL register disables the sprite.
               If enabled, data in the A and B buffers will be outputted
               whenever the beam counter equals the sprite horizontal
               position value in the SPRxPOS register.

STREQU     038 S    D     Strobe for horizontal sync with VB
                                 and EQU
STRVBL     03A S    D     Strobe for horizontal sync with VB
                                 (vertical blank)
STRHOR     03C S    D P   Strobe for horizontal sync
STRLONG    03E S    D     Strobe for identification of long
                                    horizontal line
               One of the first three strobe addresses above is
               placed on the destination address bus during the
               first refresh time slot.  The fourth strobe shown
               above is used during the second refresh time slot of
               every other line to identify lines with long counts
               (228).  There are four refresh time slots, and any
               not used for strobes will leave a null (FF) address
               on the destination address bus.

VPOSR      004 R  A       Read vertical most significant bit
                                 (and frame flop)
VPOSW      02A W  A       Write vertical most significant bit
                                 (and frame flop)
               BIT# 15,14,13,12,11,10,09,08,07,06,05,04,03,02,01,00
               USE  LOF-- -- -- -- -- -- --,-- -- -- -- -- -- -- V8
               LOF=Long frame (auto toggle control bit in BPLCON0)

VHPOSR     006 R  A       Read vertical and horizontal position of
                             beam or lightpen
VHPOSW     02C W  A       Write vertical and horizontal position
                             of beam or lightpen
               BIT# 15,14,13,12,11,10,09,08,07,06,05,04,03,02,01,00
               USE  V7 V6 V5 V4 V3 V2 V1 V0,H8 H7 H6 H5 H4 H3 H2 H1
               RESOLUTION = 1/160 of screen width (280 ns)
```

Appendix B

Register Summary—Address Order

This appendix contains information about the register set in address order.

The following codes and abbreviations are used in this appendix:

& Register used by DMA channel only.

% Register used by DMA channel usually, processors sometimes.

+ Address register pair. Low word uses DB1-DB15; high word uses DB0-DB2.

* Address not writable by the Copper.

~ Address not writable by the Copper unless COPCON is set true.

A,D,P
A=Agnus chip, D=Denise chip, P=Paula chip.

W,R
W=write; R=read,

ER Early read. This is a DMA data transfer to RAM, from either the disk or the blitter. RAM timing requires data to be on the bus earlier than microprocessor read cycles. These transfers are therefore initiated by Agnus timing, instead of a read address on the destination address bus.

S Strobe (write address with no register bits).

PTL,PTH
18-bit pointer that addresses DMA data. Must be reloaded by a processor before use (vertical blank for bit-plane and sprite pointers, and prior to starting the blitter for blitter pointers).

LCL,LCH
18-bit location (starting address) of DMA data. Used to automatically restart pointers, such as the Copper program counter (during vertical blank) and the audio sample counter (whenever the audio length count is finished).

MOD
15-bit modulo. A number that is automatically added to the memory address at the end of each line to generate the address for the beginning of the next line. This allows the blitter (or the display window) to operate on (or display) a window of data that is smaller than the actual picture in memory (memory map). Uses 15 bits, plus sign extend.

NAME	ADD	R/W	CHIP	FUNCTION
BLTDDAT	& *000	ER	A	Blitter destination early read (dummy address)
DMACONR	*002	R	A P	DMA control (and blitter status) read
VPOSR	*004	R	A	Read vert most signif. bit (and frame flop)
VHPOSR	*006	R	A	Read vert and horiz. position of beam
DSKDATR	& *008	ER	P	Disk data early read (dummy address)
JOY0DAT	*00A	R	D	Joystick-mouse 0 data (vert,horiz)
JOY1DAT	*00C	R	D	Joystick-mouse 1 data (vert,horiz)
CLXDAT	*00E	R	D	Collision data register (read and clear)
ADKCONR	*010	R	P	Audio, disk control register read
POT0DAT	*012	R	P	Pot counter pair 0 data (vert,horiz)
POT1DAT	*014	R	P	Pot counter pair 1 data (vert,horiz)
POTGOR	*016	R	P	Pot port data read (formerly POTINP)
SERDATR	*018	R	P	Serial port data and status read
DSKBYTR	*01A	R	P	Disk data byte and status read
INTENAR	*01C	R	P	Interrupt enable bits read
INTREQR	*01E	R	P	Interrupt request bits read
DSKPTH	+ *020	W	A	Disk pointer (high 3 bits)
DSKPTL	+ *022	W	A	Disk pointer (low 15 bits)
DSKLEN	*024	W	P	Disk length
DSKDAT	& *026	W	P	Disk DMA data write
REFPTR	& *028	W	A	Refresh pointer
VPOSW	*02A	W	A	Write vert most signif. bit (and frame flop)
VHPOSW	*02C	W	A	Write vert and horiz position of beam
COPCON	*02E	W	A	Coprocessor control register (CDANG)
SERDAT	*030	W	P	Serial port data and stop bits write
SERPER	*032	W	P	Serial port period and control
POTGO	*034	W	P	Pot port data write and start
JOYTEST	*036	W	D	Write to all four joystick-mouse counters at once
STREQU	& *038	S	D	Strobe for horiz sync with VB and EQU
STRVBL	& *03A	S	D	Strobe for horiz sync with VB (vert. blank)
STRHOR	& *03C	S	D P	Strobe for horiz sync
STRLONG	& *03E	S	D	Strobe for identification of long horiz. line.
BLTCON0	~040	W	A	Blitter control register 0
BLTCON1	~042	W	A	Blitter control register 1
BLTAFWM	~044	W	A	Blitter first word mask for source A
BLTALWM	~046	W	A	Blitter last word mask for source A
BLTCPTH	+ ~048	W	A	Blitter pointer to source C (high 3 bits)
BLTCPTL	+ ~04A	W	A	Blitter pointer to source C (low 15 bits)
BLTBPTH	+ ~04C	W	A	Blitter pointer to source B (high 3 bits)
BLTBPTL	+ ~04E	W	A	Blitter pointer to source B (low 15 bits)
BLTAPTH	+ ~050	W	A	Blitter pointer to source A (high 3 bits)
BLTAPTL	+ ~052	W	A	Blitter pointer to source A (low 15 bits)
BLTDPTH	+ ~054	W	A	Blitter pointer to destination D (high 3 bits)
BLTDPTL	+ ~056	W	A	Blitter pointer to destination D (low 15 bits)
BLTSIZE	~058	W	A	Blitter start and size (window width, height)
	~05A			
	~05C			
	~05E			
BLTCMOD	~060	W	A	Blitter modulo for source C
BLTBMOD	~062	W	A	Blitter modulo for source B
BLTAMOD	~064	W	A	Blitter modulo for source A
BLTDMOD	~066	W	A	Blitter modulo for destination D
	~068			
	~06A			
	~06C			
	~06E			
BLTCDAT	% ~070	W	A	Blitter source C data register
BLTBDAT	% ~072	W	A	Blitter source B data register
BLTADAT	% ~074	W	A	Blitter source A data register
	~076			
	~078			
	~07A			
	~07C			
DSKSYNC	~07E	R	P	Disk sync pattern register for disk read
COP1LCH	+ 080	W	A	Coprocessor first location register (high 3 bits)
COP1LCL	+ 082	W	A	Coprocessor first location register (low 15 bits)
COP2LCH	+ 084	W	A	Coprocessor second location register (high 3 bits)
COP2LCL	+ 086	W	A	Coprocessor second location register (low 15 bits)
COPJMP1	088	S	A	Coprocessor restart at first location
COPJMP2	08A	S	A	Coprocessor restart at second location
COPINS	08C	W	A	Coprocessor instruction fetch identify
DIWSTRT	08E	W	A	Display window start (upper left vert-horiz position)
DIWSTOP	090	W	A	Display window stop (lower right vert.-horiz. position)
DDFSTRT	092	W	A	Display bit plane data fetch start (horiz. position)
DDFSTOP	094	W	A	Display bit plane data fetch stop (horiz. position)
DMACON	096	W	A D P	DMA control write (clear or set)
CLXCON	098	W	D	Collision control
INTENA	09A	W	P	Interrupt enable bits (clear or set bits)
INTREQ	09C	W	P	Interrupt request bits (clear or set bits)
ADKCON	09E	W	P	Audio, disk, UART control
AUD0LCH	+ 0A0	W	A	Audio channel 0 location (high 3 bits)
AUD0LCL	+ 0A2	W	A	Audio channel 0 location (low 15 bits)
AUD0LEN	0A4	W	P	Audio channel 0 length
AUD0PER	0A6	W	P	Audio channel 0 period
AUD0VOL	0A8	W	P	Audio channel 0 volume
AUD0DAT	& 0AA	W	P	Audio channel 0 data
	0AC			
	0AE			
AUD1LCH	+ 0B0	W	A	Audio channel 1 location (high 3 bits)
AUD1LCL	+ 0B2	W	A	Audio channel 1 location (low 15 bits)
AUD1LEN	0B4	W	P	Audio channel 1 length
AUD1PER	0B6	W	P	Audio channel 1 period
AUD1VOL	0B8	W	P	Audio channel 1 volume
AUD1DAT	& 0BA	W	P	Audio channel 1 data
	0BC			
	0BE			

AUD2LCH	+	0C0	W	A			Audio channel 2 location (high 3 bits)
AUD2LCL	+	0C2	W	A			Audio channel 2 location (low 15 bits)
AUD2LEN		0C4	W			P	Audio channel 2 length
AUD2PER		0C6	W			P	Audio channel 2 period
AUD2VOL		0C8	W			P	Audio channel 2 volume
AUD2DAT	&	0CA	W			P	Audio channel 2 data
		0CC					
		0CE					
AUD3LCH	+	0D0	W	A			Audio channel 3 location (high 3 bits)
AUD3LCL	+	0D2	W	A			Audio channel 3 location (low 15 bits)
AUD3LEN		0D4	W			P	Audio channel 3 length
AUD3PER		0D6	W			P	Audio channel 3 period
AUD3VOL		0D8	W			P	Audio channel 3 volume
AUD3DAT	&	0DA	W			P	Audio channel 3 data
		0DC					
		0DE					
BPL1PTH	+	0E0	W	A			Bit plane 1 pointer (high 3 bits)
BPL1PTL	+	0E2	W	A			Bit plane 1 pointer (low 15 bits)
BPL2PTH	+	0E4	W	A			Bit plane 2 pointer (high 3 bits)
BPL2PTL	+	0E6	W	A			Bit plane 2 pointer (low 15 bits)
BPL3PTH	+	0E8	W	A			Bit plane 3 pointer (high 3 bits)
BPL3PTL	+	0EA	W	A			Bit plane 3 pointer (low 15 bits)
BPL4PTH	+	0EC	W	A			Bit plane 4 pointer (high 3 bits)
BPL4PTL	+	0EE	W	A			Bit plane 4 pointer (low 15 bits)
BPL5PTH	+	0F0	W	A			Bit plane 5 pointer (high 3 bits)
BPL5PTL	+	0F2	W	A			Bit plane 5 pointer (low 15 bits)
BPL6PTH	+	0F4	W	A			Bit plane 6 pointer (high 3 bits)
BPL6PTL	+	0F6	W	A			Bit plane 6 pointer (low 15 bits)
		0F8					
		0FA					
		0FC					
		0FE					
BPLCON0		100	W	A	D		Bit plane control register (misc. control bits)
BPLCON1		102	W		D		Bit plane control reg. (scroll value PF1, PF2)
BPLCON2		104	W		D		Bit plane control reg. (priority control)
		106					
BPL1MOD		108	W	A			Bit plane modulo (odd planes)
BPL2MOD		10A	W	A			Bit Plane modulo (even planes)
		10C					
		10E					
BPL1DAT	&	110	W		D		Bit plane 1 data (parallel-to-serial convert)
BPL2DAT	&	112	W		D		Bit plane 2 data (parallel-to-serial convert)
BPL3DAT	&	114	W		D		Bit plane 3 data (parallel-to-serial convert)
BPL4DAT	&	116	W		D		Bit plane 4 data (parallel-to-serial convert)
BPL5DAT	&	118	W		D		Bit plane 5 data (parallel-to-serial convert)
BPL6DAT	&	11A	W		D		Bit plane 6 data (parallel-to-serial convert)
		11C					
		11E					
SPR0PTH	+	120	W	A			Sprite 0 pointer (high 3 bits)
SPR0PTL	+	122	W	A			Sprite 0 pointer (low 15 bits)
SPR1PTH	+	124	W	A			Sprite 1 pointer (high 3 bits)
SPR1PTL	+	126	W	A			Sprite 1 pointer (low 15 bits)
SPR2PTH	+	128	W	A			Sprite 2 pointer (high 3 bits)
SPR2PTL	+	12A	W	A			Sprite 2 pointer (low 15 bits)
SPR3PTH	+	12C	W	A			Sprite 3 pointer (high 3 bits)
SPR3PTL	+	12E	W	A			Sprite 3 pointer (low 15 bits)
SPR4PTH	+	130	W	A			Sprite 4 pointer (high 3 bits)
SPR4PTL	+	132	W	A			Sprite 4 pointer (low 15 bits)
SPR5PTH	+	134	W	A			Sprite 5 pointer (high 3 bits)
SPR5PTL	+	136	W	A			Sprite 5 pointer (low 15 bits)
SPR6PTH	+	138	W	A			Sprite 6 pointer (high 3 bits)
SPR6PTL	+	13A	W	A			Sprite 6 pointer (low 15 bits)
SPR7PTH	+	13C	W	A			Sprite 7 pointer (high 3 bits)
SPR7PTL	+	13E	W	A			Sprite 7 pointer (low 15 bits)
SPR0POS	%	140	W	A	D		Sprite 0 vert-horiz start position data
SPR0CTL	%	142	W	A	D		Sprite 0 vert stop position and control data
SPR0DATA	%	144	W		D		Sprite 0 image data register A
SPR0DATB	%	146	W		D		Sprite 0 image data register B
SPR1POS	%	148	W	A	D		Sprite 1 vert-horiz start position data
SPR1CTL	%	14A	W	A	D		Sprite 1 vert stop position and control data
SPR1DATA	%	14C	W		D		Sprite 1 image data register A
SPR1DATB	%	14E	W		D		Sprite 1 image data register B
SPR2POS	%	150	W	A	D		Sprite 2 vert-horiz start position data
SPR2CTL	%	152	W	A	D		Sprite 2 vert stop position and control data
SPR2DATA	%	154	W		D		Sprite 2 image data register A
SPR2DATB	%	156	W		D		Sprite 2 image data register B
SPR3POS	%	158	W	A	D		Sprite 3 vert-horiz start position data
SPR3CTL	%	15A	W	A	D		Sprite 3 vert stop position and control data
SPR3DATA	%	15C	W		D		Sprite 3 image data register A
SPR3DATB	%	15E	W		D		Sprite 3 image data register B
SPR4POS	%	160	W	A	D		Sprite 4 vert-horiz start position data
SPR4CTL	%	162	W	A	D		Sprite 4 vert stop position and control data
SPR4DATA	%	164	W		D		Sprite 4 image data register A
SPR4DATB	%	166	W		D		Sprite 4 image data register B
SPR5POS	%	168	W	A	D		Sprite 5 vert-horiz start position data
SPR5CTL	%	16A	W	A	D		Sprite 5 vert stop position and control data
SPR5DATA	%	16C	W		D		Sprite 5 image data register A
SPR5DATB	%	16E	W		D		Sprite 5 image data register B
SPR6POS	%	170	W	A	D		Sprite 6 vert-horiz start position data
SPR6CTL	%	172	W	A	D		Sprite 6 vert stop position and control data
SPR6DATA	%	174	W		D		Sprite 6 image data register A
SPR6DATB	%	176	W		D		Sprite 6 image data register B
SPR7POS	%	178	W	A	D		Sprite 7 vert-horiz start position data
SPR7CTL	%	17A	W	A	D		Sprite 7 vert stop position and control data
SPR7DATA	%	17C	W		D		Sprite 7 image data register A
SPR7DATB	%	17E	W		D		Sprite 7 image data register B

COLOR00	180	W	D	Color table 00
COLOR01	182	W	D	Color table 01
COLOR02	184	W	D	Color table 02
COLOR03	186	W	D	Color table 03
COLOR04	188	W	D	Color table 04
COLOR05	18A	W	D	Color table 05
COLOR06	18C	W	D	Color table 06
COLOR07	18E	W	D	Color table 07
COLOR08	190	W	D	Color table 08
COLOR09	192	W	D	Color table 09
COLOR10	194	W	D	Color table 10
COLOR11	196	W	D	Color table 11
COLOR12	198	W	D	Color table 12
COLOR13	19A	W	D	Color table 13
COLOR14	19C	W	D	Color table 14
COLOR15	19E	W	D	Color table 15
COLOR16	1A0	W	D	Color table 16
COLOR17	1A2	W	D	Color table 17
COLOR18	1A4	W	D	Color table 18
COLOR19	1A6	W	D	Color table 19
COLOR20	1A8	W	D	Color table 20
COLOR21	1AA	W	D	Color table 21
COLOR22	1AC	W	D	Color table 22
COLOR23	1AE	W	D	Color table 23
COLOR24	1B0	W	D	Color table 24
COLOR25	1B2	W	D	Color table 25
COLOR26	1B4	W	D	Color table 26
COLOR27	1B6	W	D	Color table 27
COLOR28	1B8	W	D	Color table 28
COLOR29	1BA	W	D	Color table 29
COLOR30	1BC	W	D	Color table 30
COLOR31	1BE	W	D	Color table 31
RESERVED	1110X			
RESERVED	1111X			
NO-OP (NULL)	1FE			

Appendix C

Custom Chip Pin Allocation List

NOTE: * Means an active low signal.

AGNUS PIN ASSIGNMENT

PIN #	DESIGNATION	FUNCTION	DEFINITION
1	D8	DATA BUS 8	I/O
2	D7	DATA BUS 7	I/O
3	D6	DATA BUS 6	I/O
4	D5	DATA BUS 5	I/O
5	D4	DATA BUS 4	I/O
6	D3	DATA BUS 3	I/O
7	D2	DATA BUS 2	I/O
8	D1	DATA BUS 1	I/O
9	D0	DATA BUS 0	I/O
10	VCC	+5 VOLT	I
11	RES*	SYSTEM RESET	I
12	INT3*	INTERRUPT LEVEL 3	O
13	DMAL	DMA REQUEST LINE	I
14	BLS*	BLITTER SLOWDOWN	I
15	DBR*	DATA BUS REQUEST	O
16	ARW*	AGNUS RAM WRITE	O
17	RGA8	REGISTER ADDRESS 8	I/O
18	RGA7	REGISTER ADDRESS 7	I/O
19	RGA6	REGISTER ADDRESS 6	I/O
20	RGA5	REGISTER ADDRESS 5	I/O
21	RGA4	REGISTER ADDRESS 4	I/O
22	RGA3	REGISTER ADDRESS 3	I/O
23	RGA2	REGISTER ADDRESS 2	I/O
24	RGA1	REGISTER ADDRESS 1	I/O
25	CCK	COLOR CLOCK	I
26	CCKQ	COLOR CLOCK DELAY	I
27	VSS	GROUND	I
28	DRA0	DYNAMIC RAM ADDRESS 0	O
29	DRA1	DYNAMIC RAM ADDRESS 1	O
30	DRA2	DYNAMIC RAM ADDRESS 2	O
31	DRA3	DYNAMIC RAM ADDRESS 3	O
32	DRA4	DYNAMIC RAM ADDRESS 4	O
33	DRA5	DYNAMIC RAM ADDRESS 5	O
34	DRA6	DYNAMIC RAM ADDRESS 6	O
35	DRA7	DYNAMIC RAM ADDRESS 7	O
36	DRA8	DYNAMIC RAM ADDRESS 8	O
37	LP*	LIGHT PEN INPUT	I
38	VSY*	VERTICAL SYNC	I/O
39	CSY*	COMPOSITE SYNC	O
40	HSY*	HORIZONTAL SYNC	I/O
41	VSS	GROUND	I
42	D15	DATA BUS 15	I/O
43	D14	DATA BUS 14	I/O
44	D13	DATA BUS 13	I/O
45	D12	DATA BUS 12	I/O
46	D11	DATA BUS 11	I/O
47	D10	DATA BUS 10	I/O
48	D9	DATA BUS 9	I/O

DENISE PIN ASSIGNMENT

PIN #	DESIGNATION	FUNCTION	DEFINITION
1	D6	DATA BUS 6	I/O
2	D5	DATA BUS 5	I/O
3	D4	DATA BUS 4	I/O
4	D3	DATA BUS 3	I/O
5	D2	DATA BUS 2	I/O
6	D1	DATA BUS 1	I/O
7	D0	DATA BUS 0	I/O
8	M1H	MOUSE 1 HORIZONTAL	I
9	M0H	MOUSE 0 HORIZONTAL	I
10	RGA8	REGISTER ADDRESS 8	I
11	RGA7	REGISTER ADDRESS 7	I
12	RGA6	REGISTER ADDRESS 6	I
13	RGA5	REGISTER ADDRESS 5	I
14	RGA4	REGISTER ADDRESS 4	I
15	RGA3	REGISTER ADDRESS 3	I
16	RGA2	REGISTER ADDRESS 2	I
17	RGA1	REGISTER ADDRESS 1	I
18	BURST*	COLOR BURST	O
19	VCC	+5 VOLT	I
20	R0	VIDEO RED BIT 0	O
21	R1	VIDEO RED BIT 1	O
22	R2	VIDEO RED BIT 2	O
23	R3	VIDEO RED BIT 3	O
24	B0	VIDEO BLUE BIT 0	O
25	B1	VIDEO BLUE BIT 1	O
26	B2	VIDEO BLUE BIT 2	O
27	B3	VIDEO BLUE BIT 3	O
28	G0	VIDEO GREEN BIT 0	O
29	G1	VIDEO GREEN BIT 1	O
30	G2	VIDEO GREEN BIT 2	O
31	G3	VIDEO GREEN BIT 3	O
32	N/C	NOT CONNECTED	N/C
33	ZD*	BACKGROUND INDICATOR	O
34	N/C	NOT CONNECTED	N/C
35	7M	7.15909 MHZ	I
36	CCK	COLOR CLOCK	I
37	VSS	GROUND	I
38	M0V	MOUSE 0 VERTICAL	I
39	M1V	MOUSE 1 VERTICAL	I
40	D15	DATA BUS 15	I/O
41	D14	DATA BUS 14	I/O
42	D13	DATA BUS 13	I/O
43	D12	DATA BUS 12	I/O
44	D11	DATA BUS 11	I/O
45	D10	DATA BUS 10	I/O
46	D9	DATA BUS 9	I/O
47	D8	DATA BUS 8	I/O
48	D7	DATA BUS 7	I/O

PAULA PIN ASSIGNMENT

PIN #	DESIGNATION	FUNCTION	DEFINITION
1	D8	DATA BUS 8	I/O
2	D7	DATA BUS 7	I/O
3	D6	DATA BUS 6	I/O
4	D5	DATA BUS 5	I/O
5	D4	DATA BUS 4	I/O
6	D3	DATA BUS 3	I/O
7	D2	DATA BUS 2	I/O
8	VSS	GROUND	I
9	D1	DATA BUS 1	I/O
10	D0	DATA BUS 0	I/O
11	RES*	SYSTEM RESET	I
12	DMAL	DMA REQUEST LINE	O
13	IPL0*	INTERRUPT LINE 0	O
14	IPL1*	INTERRUPT LINE 1	O
15	IPL2*	INTERRUPT LINE 2	O
16	INT2*	INTERRUPT LEVEL 2	I
17	INT3*	INTERRUPT LEVEL 3	I
18	INT6*	INTERRUPT LEVEL 6	I
19	RGA8	REGISTER ADDRESS 8	I
20	RGA7	REGISTER ADDRESS 7	I
21	RGA6	REGISTER ADDRESS 6	I
22	RGA5	REGISTER ADDRESS 5	I
23	RGA4	REGISTER ADDRESS 4	I
24	RGA3	REGISTER ADDRESS 3	I
25	RGA2	REGISTER ADDRESS 2	I
26	RGA1	REGISTER ADDRESS 1	I
27	VCC	+5 VOLT	I
28	CCK	COLOR CLOCK	I
29	CCKQ	COLOR CLOCK DELAY	I
30	AUDB	RIGHT AUDIO	O
31	AUDA	LEFT AUDIO	O
32	POT0X	POT 0X	I/O
33	POT0Y	POT 0Y	I/O
34	VSSANA	ANALOG GROUND	I
35	POT1X	POT 1X	I/O
36	POT1Y	POT 1Y	I/O
37	DKRD*	DISK READ DATA	I
38	DKWD*	DISK WRITE DATA	O
39	DKWE	DISK WRITE ENABLE	O
40	TXD	SERIAL TRANSMIT DATA	O
41	RXD	SERIAL RECEIVE DATA	I
42	D15	DATA BUS 15	I/O
43	D14	DATA BUS 14	I/O
44	D13	DATA BUS 13	I/O
45	D12	DATA BUS 12	I/O
46	D11	DATA BUS 11	I/O
47	D10	DATA BUS 10	I/O
48	D9	DATA BUS 9	I/O

Appendix D

System Memory Map

```
ADDRESS RANGE  NOTES
-------------  --------------------------------------

000000-03FFFF  256k bytes of RAM

040000-07FFFF  256k bytes of display RAM (option card)

080000-1FFFFF  Do not use

200000-9FFFFF  External expansion space

A00000-BEFFFF  Do not use

BFD000-BFDF00  8520-B (access only at EVEN byte addresses)
   =      =

BFE001-BFEF01  8520-A (access only at ODD byte addresses)
   =      =

          The  underlined digit chooses which of the
          16 internal registers of the 8520 is to be
          accessed.

          Register names are given below.

C00000-DFEFFF          Reserved for future use

DFF000-DFFFFF          Special purpose chips, where
                       the last three digits specify
                       the chip register WORD address.

                       The chip addresses are specified
                       in separate pages immediately
                       following this overall memory map.

E00000-E7FFFF          Reserved for future use - do not use

E80000-EFFFFF          Expansion slot decoding

F00000-F7FFFF          Reserved - do not use

F80000-FFFFFF          System ROM

DEVELOPMENT SYSTEM ROMs located at start address FE0000

FINAL SYSTEM ROMs will probably be located at FC0000
```

The names of the registers within the 8520s are as follows. The address at which each is to be accessed is given in this list.

Address for:

8520-A	8520-B	NAME	EXPLANATION (write)/(read mode)
BFE001	BFD000	PRA	Peripheral data register A
BFE101	BFD100	PRB	Peripheral data register B
BFE201	BFD200	DDRB	Data direction register A
BFE301	BFD300	DDRA	Data direction register B
BFE401	BFD400	TALO	TIMER A low register
BFE501	BFD500	TAHI	TIMER A high register
BFE601	BFD600	TBLO	TIMER B low register
BFE701	BFD700	TBHI	TIMER B high register
BFE801	BFD800		Event LSB
BFE901	BFD900		Event 8 - 15
BFEA01	BFDA00		Event MSB
BFEB01	BFDB00		No connect
BFEC01	BFDC00	SDR	Serial data register
BFED01	BFDD00	ICR	Interrupt control register
BFEE01	BFDE00	CRA	Control register A
BFEF01	BFDF00	CRB	Control register B

Appendix E

Interfaces

This appendix consists of four distinct parts, related to the way in which the Amiga talks to the outside world.

The first part specifies the pinouts of the externally accessible connectors and the power available at each connector. It does not, however, provide timing or loading information.

The second part briefly describes the functions of those pins whose purpose may not be evident.

The third part contains a list of the connections for certain internal connectors, notably the disk.

The fourth part specifies how various signals relate to the available ports of the 8520. This information enables the programmer to relate the port addresses to the outside-world items (or internal control signals) that are to be affected. The third and fourth parts are primarily for the use of the systems programmer and should generally not be utilized by applications programmers. Systems software normally is configured to handle the setting of particular signals, no matter how the physical connections may change. In other words, if you have a version of the system software that matches the revision level of the machine (normally a true condition), when you ask that a particular bit be set, you don't care which port that bit is connected to. Thus, applications programmers should rely on system documentation instead of going directly to the ports. Note also that in a multitasking operating system, many different tasks may be competing for the use of the system resources. Applications programmers should follow the established rules for resource access in order to assure compatibility of their software with the system.

See the figures at the end of this appendix for more information about the fire buttons, light pen, mouse, and the "pot" counters.

************** PART 1 - OUTSIDE WORLD CONNECTORS ********************

This is a list of the connections to the outside world on the Amiga.

SERIAL COM ...DB25 FEMALE (J6)

(The center column is the AMIGA connection, the others are specified in this table merely to show how the AMIGA RS-232-C connection compares to other defined interconnect methods.)

PIN	RS232	AMIGA	HAYES	DESCRIPTION
1	GND	GND		FRAME GROUND
2	TXD	TXD	TXD	TRANSMIT DATA
3	RXD	RXD	RXD	RECEIVE DATA
4	RTS	RTS		REQUEST TO SEND
5	CTS	CTS	CTS	CLEAR TO SEND
6	DSR	DSR	DSR	DATA SET READY
7	GND	GND	GND	SYSTEM GROUND
8	CD	CD	CD	CARRIER DETECT
9				
10				
11				
12	S.SD		SI	
13	S.CTS			
14	S.TXD	-5		- 5 VOLT POWER
15	TXC	AUDO		AUDIO OUT OF AMIGA
16	S.RXD	AUDI		AUDIO IN TO AMIGA
17	RXC	EB		BUFFERED PORT CLOCK 716kHz
18		INT2*		INTERRUPT LINE TO AMIGA
19	S.RTS			
20	DTR	DTR	DTR	DATA TERMINAL READY
21	SQD	+5		+ 5 VOLT POWER
22	RI		RI	
23	SS	+12		+12 volt power
24	TXC1	C2*		3.58 MHZ CLOCK
25		RESB*		BUFFERED SYSTEM RESET

PARALLEL COM ...DB25 MALE (J8)

1	DRDY*	14	GND
2	D0	15	GND
3	D1	16	GND
4	D2	17	GND
5	D3	18	GND
6	D4	19	GND
7	D5	20	GND
8	D6	21	GND
9	D7	22	GND
10	ACK*	23	+ 5
11	BUSY (data)	24	
12	POUT (clk)	25	RESET*
13	SEL		

KEYBOARD ...RJ11 (J9)

1	+5
2	CLOCK
3	DATA
4	GND

RGB ...DB23 MALE (J3)

1	XCLK*	13	GNDRTN (Return for XCLKEN*)
2	XCLKEN*	14	ZD*
3	RED	15	C1*
4	GREEN	16	GND
5	BLUE	17	GND
6	DI	18	GND
7	DB	19	GND
8	DG	20	GND
9	DR	21	-5 VOLT POWER
10	CSYNC*	22	+12 VOLT POWER
11	HSYNC*	23	+5 VOLT POWER
12	VSYNC*		

TV VIDEO ...8 PIN DIN (J2)

1	N.C.
2	GND
3	AUDIO LEFT
4	COMP VIDEO
5	GND
6	N.C.
7	+12 VOLT POWER
8	AUDIO RIGHT

DISK EXTERNAL ...DB23 FEMALE (J7)

1	RDY*	13	SIDEB*
2	DKRD*	14	WPRO*
3	GND	15	TK0*
4	GND	16	DKWEB*
5	GND	17	DKWDB*
6	GND	18	STEPB*
7	GND	19	DIRB
8	MTRXD*	20	SEL3B*
9	SEL2B*	21	SEL1B*
10	DRESB*	22	INDEX*
11	CHNG*	23	+12
12	+5		

RAMEX ...60 PIN EDGE (.156) (P1)

1	gnd	A	gnd
2	D15	B	D14
3	+5	C	+5
4	D12	D	D13
5	gnd	E	gnd
6	D11	F	D10
7	+5	H	+5
8	D8	J	D9
9	gnd	K	gnd
10	D7	L	D6
11	+5	M	+5
12	D4	N	D5
13	gnd	P	gnd
14	D3	R	D2
15	+5	S	+5
16	D0	T	D1
17	gnd	U	gnd
18	DRA4	V	DRA3
19	DRA5	W	DRA2
20	DRA6	X	DRA1
21	DRA7	Y	DRA0
22	gnd	Z	gnd
23	RAS*	AA	RRW*
24	gnd	BB	gnd
25	gnd	CC	gnd
26	CASU0*	DD	CASU1*
27	gnd	EE	gnd
28	CASL0*	FF	CASL1*
29	+5	HH	+5
30	+5	JJ	+5

EXPANSION ...86 PIN EDGE (.1) (P2)

1	gnd	44	IPL2*
2	gnd	45	A16
3	gnd	46	BERR*
4	gnd	47	A17
5	+5	48	VPA*
6	+5	49	gnd
7	exp	50	E
8	-5	51	VMA*
9	exp	52	A18
10	+12	53	RES*
11	exp	54	A19
12	CONFIG	55	HLT*
13	gnd	56	A20
14	C3*	57	A22
15	CDAC	58	A21
16	C1*	59	A23
17	OVR*	60	BR*
18	XRDY	61	gnd
19	INT2*	62	BGACK*
20	PALOPE*	63	PD15
21	A5	64	BG*
22	INT6*	65	PD14
23	A6	66	DTACK*
24	A4	67	PD13
25	gnd	68	PRW*
26	A3	69	PD12
27	A2	70	LDS*
28	A7	71	PD11
29	A1	72	UDS*
30	A8	73	gnd
31	FC0	74	AS*
32	A9	75	PD0
33	FC1	76	PD10
34	A10	77	PD1
35	FC2	78	PD9
36	A11	79	PD2
37	gnd	80	PD8
38	A12	81	PD3
39	A13	82	PD7
40	IPL0*	83	PD4
41	A14	84	PD6
42	IPL1*	85	gnd
43	A15	86	PD5

POWER ...7 PIN STRAIGHT (.156) (J14)

1 -5
2 +12
3 gnd
4 gnd
5 +5
6 +5
7 tick

JOY STICKS ...DB9 male (J11 = right J12 = left)

1 FORWARD* (MOUSE V)
2 BACK* (MOUSE H)
3 LEFT* (MOUSE VQ)
4 RIGHT* (MOUSE HQ)
5 POT X (or button 3 ... if used)
6 FIRE* (or button 1)
7 +5
8 GND
9 POT Y (or button 2)

The following port power allocation list is based on many things, including known peripheral requirements and existing power supply capabilities. These numbers are maximums for each port when used independently, but the numbers can be accumulated (except for joysticks) when a particular system configuration will guarantee that it exclusively uses more than one port.

The power pins of both joystick ports are tied together and to a current limited +5 supply. At present, the current limit is set at 700 ma peak with a 400 ma foldback at steady state short circuit conditions. The combined utilization of both ports is limited to 250 ma to insure a minimum voltage drop at the pins.

PORT	+5(ma)	+12(ma)	-5(ma)
RF modulator	.	60	.
RGB	300	175	50
Serial	100	50	50
External disk	270	160	.
Parallel	100	.	.
Expansion	1000	50	50
Joystick 0	125	.	.
Joystick 1	125	.	.

************** PART 2 - MORE OUTSIDE WORLD ********************

PARALLEL INTERFACE CONNECTOR SPECIFICATION

The 25-pin D-type connector with pins (DB25P=male) at the rear of the Amiga is nominally used to interface to parallel printers. In this capacity, data flows from the Amiga to the printer. This interface may also be used for input or bidirectional data transfers. The implementation is similar to Centronics, but the pin assignment and drive characteristics vary significantly from that specification (see Pin Assignment). Signal names correspond to those used in the other places in this appendix, when possible.

PARALLEL CONNECTOR PIN ASSIGNMENT (J8)

PIN	NAME	DIR	NOTES
1	DRDY*	O	Output-data-ready signal to parallel device in output mode, used in conjunction with ACK* (pin 10) for a two-line asynchronous handshake. Functions as input data accepted from Amiga in input mode (similar to ACK* in output mode). See timing diagrams in the following section.
2	D0	I/O	
3	D1	I/O	
4	D2	I/O	
5	D3	I/O	D0-D7 comprise an eight-bit bidirectional bus for communication with parallel devices, nominally, a printer.
6	D4	I/O	
7	D5	I/O	
8	D6	I/O	
9	D7	I/O	
10	ACK*	I	Output-data-acknowledge from parallel device in output mode, used in conjunction with DRDY* (pin 1) for a two-line asynchronous handshake. Functions as input-data-ready from parallel device in input mode (similar to DRDY* in output mode). See timing diagrams. The 8520 can be programmed to conditionally generate a level 2 interrupt to the 68000 whenever the ACK* input goes active.
11	BUSY	I/O	This is a general purpose I/O pin shorted to a serial data I/O pin (serial clock on pin 12). Note: Nominally used to indicate printer buffer full.
12	POUT	I/O	This is a general purpose I/O pin shorted to a serial clock I/O pin (serial data on pin 11). Note: Nominally used to indicate printer paper out.
13	SEL	I/O	This is a general purpose I/O pin. Note: nominally a select output from the parallel device to the Amiga.
14	GND		
15	GND		
16	GND		
17	GND		
18	GND		
19	GND		
20	GND		

21	GND		
22	GND		
23	+5V		100 ma maximum. *** WARNING +5V ***
24	---		
25	RESET*	O	Amiga system reset

PARALLEL CONNECTOR INTERFACE TIMING, OUTPUT CYCLE

```
PA<7:0>___ ______________________________________________ __
PB<7:0>___X_____________________________________________X__
          |<-- T1 --->|                                 |
                      |           |<-------- T2 ------>|
       _______________V           V_______________________
DRDY*                 |___________|
  Output data ready   |<- T3 ->|
                      |<--- T4 --->|
       ____________________________|<- T5 -->|__________
ACK*                               |_________|
  Output data acknowledge
```

Microseconds Min Typ Max	
T1: 4.3 -x- 5.3	Output data setup to ready delay
T2: nsp -x- upc	Output data hold time.
T3: nsp 1:4 nsp	Output data ready width.
T4: 0 -x- upc	Ready to acknowledge delay.
T5: nsp -x- upc	Acknowledge width.

nsp = not specified
upc = under program control

PARALLEL CONNECTOR INTERFACE TIMING, INPUT CYCLE

```
PA<7:0>___ ______________________________________________ __
PB<7:0>___X_____________________________________________X__
          |<-- T1 --->|
                      |                  T2 -->|<----->|.
       _______________V           ____________|___________
ACK*                  |___________|            |
  Input data ready    |<- T3 ->|               |
                      |<-- T4 --->|
       ___________________________|<- T5 -->|___________
DRDY*                             |_________|
  Input data acknowledge
```

Microseconds Min Typ Max	
T1: 0 -x- upc	Input data setup time.
T2: nsp -x- upc	Input data hold time.
T3: nsp -x- upc	Input data ready width.
T4: upc -x- upc	Input data ready to data acknowledge delay.
T5: nsp 1.4 nsp	Input data acknowledge width

nsp = not specified
upc = under program control

SERIAL INTERFACE CONNECTOR SPECIFICATION

This 25-pin D-type connector with sockets (DB25S=female) is used to interface to RS-232-C standard signals. Signal names correspond to those used in other places in this appendix, when possible.

WARNING: Pins 14, 21 and 23 carry power. Do not connect to these pins inadvertently because they can permanently damage external equipment. Also, pins 15-16, 23-25 carry non-standard signals and should not be connected. NEVER use a fully wired 25 line cable.

SERIAL INTERFACE CONNECTOR PIN ASSIGNMENT (J6)

PIN	NAME	DIR	RS-232-C STD	NOTES
1	FGND		y	Frame ground -- do not tie to logic ground
2	TXD	O	y	Transmit data
3	RXD	I	y	Receive data
4	RTS	O	y	Request to send
5	CTS	I	y	Clear to send
6	DSR	I	y	Data set ready
7	GND		y	Signal ground -- do not tie to frame ground
8	CD	I	y	Carrier detect
9	---		n	
10	---		n	
11	---		y	
12	---		n	
13	---		n	
14	-5V		n*	50 ma maximum *** WARNING -5V ***
15	AUDO	O	n*	Audio output from left (channels 0, 3) port intended to send audio to the modem.
16	AUDI	I	n*	Audio input to right (channels 1, 2) port, intended to receive audio from the modem; this input is mixed with the analog output of the right (channels 1, 2). It is not digitized or used by the computer in any way.
17	EB	O	n*	716 KHz clock that supports 68000 peripheral transfers, intended for modem interface; this is the buffered version of the E clock from the 68000.
18	INT2*	I	n*	Asserting this OPEN COLLECTOR signal will generate a level 2 interrupt to the 68000 if it is enabled
19	---		n	
20	DTR	O	y	Data terminal ready.
21	+5V		n*	100 ma maximum *** WARNING +5V ***
22	---		n	
23	+12V		n*	50 ma maximum *** WARNING +12V ***
24	C2*	O	n*	3.58 MHz intended for modems that need a colorburst clock.

25 RESB* O n* Amiga system reset.

n*: See warning above

SERIAL INTERFACE CONNECTOR TIMING

Maximum operating frequency is 19.2 KHz. Refer to EIA standard RS-232-C for operating and installation specifications.
A rate of 31.25 KHz will be supported through the use of a MIDI adapter

Modem control signals (CTS, RTS, DTR, DSR, CD) are completely under software control. The modem control lines have no hardware affect on and are completely asynchronous to TXD and RXD.

SERIAL INTERFACE CONNECTOR ELECTRICAL CHARACTERISTICS

OUTPUTS	MIN	TYP	MAX		
Vo(-):	-2.5	-x-	-5.5	V	Negative output voltage range
Vo(+):	8	-x-	13.2	V	Positive output voltage range
Io:		-x-	10	ma	Output current

INPUTS	MIN	TYP	MAX		
Vi(+):	3	-x-	25	V	Positive input voltage range
Vi(-):	-25	-x-	.5	V	Negative input voltage range
Vhys:	-x-	1	-x-	V	Input hysteresis voltage
Ii:	.3	-x-	10	ma	Input current

Unconnected inputs are interpreted the same as positive input voltages.

GAME CONTROLLER INTERFACE CONNECTOR SPECIFICATION

The two 9-pin D-type connectors with pins (male) at the right of the Amiga nearer the front are used to interface to four types of devices.

1. Mouse or trackball, 3 buttons max.
2. Digital joystick, 2 buttons max.
3. Proportional (pot or proportional joystick), 2 buttons max.
4. Light pen, including pen-pressed-to-screen button.

The connector pin assignments are discussed in sections organized by similar hardware and/or software operating requirements as shown in the previous list. Signal names follow those used elsewhere in this appendix, when possible.

J11 is the right controller port connector (JOY1DAT, POT1DAT).
J12 is the left controller port connector (JOY0DAT, POT0DAT).

NOTE: While most of the hardware discussed below is directly accessible, hardware should be accessed through ROM kernel software This will keep future hardware changes transparent to the user.

GAME CONTROLLER INTERFACE TO MOUSE/TRACKBALL QUADRATURE INPUTS

A mouse or trackball is a device that translates planar motion into pulse trains. Quadrature techniques are employed to preserve the direction as well as magnitude of displacement. The registers JOY0DAT and JOY1DAT become counter registers, with y displacement in the high byte and x in the low byte. Movement causes the following action.

Up. y decrements
Down: y increments
Right: x increments
Left. x decrements

To determine displacement, JOYxDAT is read twice with corresponding x and y values subtracted (careful, modulo 128 arithmetic). Note that if either count changes by more than 127, both distance and direction become ambiguous. There is a relationship between the sampling interval and the maximum speed (that is, change in distance) that can be resolved as follows:

Velocity < Distance(max) / SampleTime

Velocity < SQRT(DeltaX**2 + DeltaY**2) / SampleTime

For an Amiga with a 200 count-per-inch mouse sampling during each vertical blanking interval, the maximum velocity in either the X or Y direction becomes:

Velocity < (128 Counts * 1 inch/200 Counts) / .017 sec = 38 in/sec

which should be sufficient for most users.

NOTE: The Amiga software is designed to do mouse update cycles during vertical blanking. The horizontal and vertical counters are always valid and may be read at any time.

CONNECTOR PIN USAGE FOR MOUSE/TRACKBALL QUADRATURE INPUTS

PIN	MNEMONIC	DESCRIPTION	HARDWARE REGISTER/NOTES
1	V	Vertical pulses	JOY[0/1]DAT<15:8>
2	H	Horizontal pulses	JOY[0/1]DAT(7:0>
3	VQ	Vertical quadrature pulses	JOY[0/1]DAT<15:8>
4	HQ	Horizontal quadrature pulses	JOY[0/1]DAT<7:0>
5	UBUT*	Unused mouse button	See Proportional Inputs
6	LBUT*	Left mouse button	See Fire Button.
7	+5V	125ma max, 200ma surge	Total both ports.
8	Ground		
9	RBUT*	Right mouse button	See Proportional Inputs

GAME PORT INTERFACE TO DIGITAL JOYSTICKS

A joystick is a device with four normally opened switches arranged 90 degrees apart. The JOY[0/1]DAT registers become encoded switch input ports as follows:

Forward: bit#9 xor bit#8
Left: bit#9
Back: bit#1 xor bit#0
Right: bit#1

Data is encoded to facilitate the mouse/trackball operating mode.

NOTE: The right and left direction inputs are also designed to be right and left buttons, respectively, for use with proportional inputs. In this case, the forward and back inputs are not used, while right and left become button inputs rather than joystick inputs

The JOY[0/1]DAT registers are always valid and may be read at any time.

CONNECTOR PIN USAGE FOR DIGITAL JOYSTICK INPUTS

PIN	MNEMONIC	DESCRIPTION	HARDWARE REGISTER/NOTES
1	FORWARD*	Forward joystick switch	JOY[0/1]DAT<9 xor 8>
2	BACK*	Back joystick switch	JOY[0/1]DAT(1 xor 0>
3	LEFT*	Left joystick switch	JOY[0/1]DAT<9>
4	RIGHT*	Right joystick switch	JOY[0/1]DAT<1>
5	Unused		
6	FIRE*	Left mouse button	See Fire Button.
7	+5V	125ma max, 200ma surge	Total both ports.
8	Ground		
9	Unused		

GAME PORT INTERFACE TO FIRE BUTTONS

The fire buttons are normally opened switches routed to the 8520 adapter PRA0 as follows:

PRA0 bit 7 = Fire* left controller port
PRA0 bit 6 = Fire* right controller port

Before reading this register, the corresponding bits of the data direction register must be cleared to define input mode:

DDRA0<7:6> cleared as appropriate

NOTE: Do not disturb the settings of other bits in DDRA0 (Use of ROM kernel calls is recommended).

Fire buttons are always valid and may be read at any time.

CONNECTOR PIN USAGE FOR FIRE BUTTON INPUTS

PIN	MNEMONIC	DESCRIPTION
1	-x-	
2	-x-	
3	-x-	
4	-x-	
5	-x-	
6	FIRE*	Left mouse button/fire button
7	-x-	
8	-x-	
9	-x-	

GAME PORT INTERFACE TO PROPORTIONAL CONTROLLERS

Resistive (potentiometer) element linear taper proportional controllers are supported up to 528k Ohms max (470k +/- 10% recommended). The JOY[0/1]DAT registers contain digital translation values for y in the high byte and x in the low byte. A higher count value indicates a higher external resistance. The Amiga performs an integrating analog-to-digital conversion as follows:

1. POT[0/1]DAT registers are reset and the analog input capacitors are discharged for the first 7 (261 lines) or 8 (262 lines) horizontal lines.
2. Once per horizontal line, each analog input is compared to an internal reference.
3. Any counter whose analog input exceeds the reference stops incrementing. The counter is stopped for the duration of the vertical frame.
4. Any counter whose analog input is less than the reference continues to increment.

NOTE: The POTY and POTX inputs are designated as "right mouse button" and "unused mouse button" respectively. An opened switch corresponds to high resistance, a closed switch to a low resistance. The buttons are also available in POTGO and POTINP registers. It is recommended that ROM kernel calls be used for future hardware compatibility.

The POT[0/1]DAT registers are typically read during video blanking, but MAY be available prior to that.

CONNECTOR PIN USAGE FOR PROPORTIONAL INPUTS

PIN	MNEMONIC	DESCRIPTION	HARDWARE REGISTER/NOTES
1	Unused		
2	Unused		
3	LBUT*	Left button	See Digital Joystick
4	RBUT*	Right button	See Digital Joystick
5	POTX	X analog in	POT[0/1]DAT<7:0>, POTGO, POTINP
6	Unused		
7	+5V	125ma max, 200 ma surge	
8	Ground		
9	POTY	Y analog in	POT[0,1]DAT<15:8>, POTGO, POTINP

GAME PORT INTERFACE TO LIGHT PEN

A light pen is an optoelectronic device whose light-sensitive portion is placed in proximity to a CRT. As the electron beam sweeps past the light pen, a trigger pulse is generated which can be enabled to latch the horizontal and vertical beam positions. There is no hardware bit to indicate this trigger, but this can be determined in the two ways as shown in chapter 8, "Interface Hardware."

Light pen position is usually read during blanking, but MAY be available prior to that.

CONNECTOR PIN USAGE FOR LIGHT PEN INPUTS

PIN	MNEMONIC	DESCRIPTION	HARDWARE REGISTER/NOTES
1	Unused		
2	Unused		
3	Unused		
4	Unused		
5	LPENPR*	Light pen pressed	See Proportional Inputs
6	LPENTG*	Light pen trigger	VPOSR, VHPOSR
7	+5V	125ma max, 200 ma surge	Both ports
8	Ground		
9	Unused		

EXTERNAL DISK INTERFACE CONNECTOR SPECIFICATION

The 23-pin D-type connector with sockets (DB23S) at the rear of the Amiga is nominally used to interface to MFM devices.

EXTERNAL DISK CONNECTOR PIN ASSIGNMENT (J7)

PIN	NAME	DIR	NOTES
1	RDY*	I/O	If motor on, indicates disk installed and up to speed. If motor not on, identification mode. See below.
2	DKRD*	I	MFM input data to Amiga.
3	GND		
4	GND		
5	GND		
6	GND		
7	GND		
8	MTRXD*	OC	Motor on data, clocked into drive's motor-on flip-flop by the active transition of SELxB*. Guaranteed setup time is 1.4 usec. Guaranteed hold time is 1.4 usec.
9	SEL2B*	OC	Select drive 2
10	DRESB*	OC	Amiga system reset. Drives should reset their motor-on flip-flops and set their write-protect flip-flops.
11	CHNG*	I/O	Note: Nominally used as an open collector input. Drive's change flop is set at power up or when no disk is not installed. Flop is reset when drive is selected and the head stepped, but only if a disk is installed.
12	+5V		270 ma maximum; 410 ma surge When below 3.75V, drives are required to reset their motor-on flops, and set their write-protect flops.
13	SIDEB*	O	Side 1 if active, side 0 if inactive
14	WPRO*	I/O	Asserted by selected, write-protected disk.
15	TK0*	I/O	Asserted by selected drive when read/write head is positioned over track 0.
16	DKWEB*	OC	Write gate (enable) to drive.
17	DKWDB*	OC	MFM output data from Amiga.
18	STEPB*	OC	Selected drive steps one cylinder in the direction indicated by DIRB.
19	DIRB	OC	Direction to step the head. Inactive to step towards center of disk (higher-numbered tracks).
20	SEL3B*	OC	Select drive 3.
21	SEL1B*	OC	Select drive 1.
22	INDEX*	I/O	Index is a pulse generated once per disk revolution, between the end and beginning of cylinders. The 8520 can be programmed to conditionally generate a level 6 interrupt to the 68000 whenever the INDEX* input goes active.
23	+12V		160 ma maximum; 540 ma surge.

EXTERNAL DISK CONNECTOR IDENTIFICATION MODE

An identification mode is provided for reading a 32-bit serial identification data stream from an external device. To initialize this mode, the motor must be turned on, then off. See pin 8 MTRXD* for a discussion of how to turn the motor on and off. The transition from motor on to motor off reinitializes the serial shift register.
After initialization, the SELxB* signal should be left in the inactive state.
Now enter a loop where SELxB* is driven active, read serial input data on RDY* (pin 1), and drive SELxB* inactive. Repeat this loop a total of 32 times to read in 32 bits of data. The most significant bit is received first.

EXTERNAL DISK CONNECTOR DEFINED IDENTIFICATIONS

$0000 0000 - no drive present.
$FFFF FFFF - Amiga standard 3.25 diskette
$5555 5555 - 48 TPI double-density, double-sided.

As with other peripheral ID's, users should contact Commodore-Amiga for ID assignment.
The serial input data is active low and must therefore be inverted to be consistent with the above table.

EXTERNAL DISK CONNECTOR LIMITATIONS

1. The total cable length, including daisy chaining, must not exceed 1 meter.

2. A maximum of 3 external devices may reside on this interface.

3. Each device must provide a 1000-Ohm pull-up resistor on those outputs driven by an open-collector device on the Amiga (pins 8-10, 16-21).

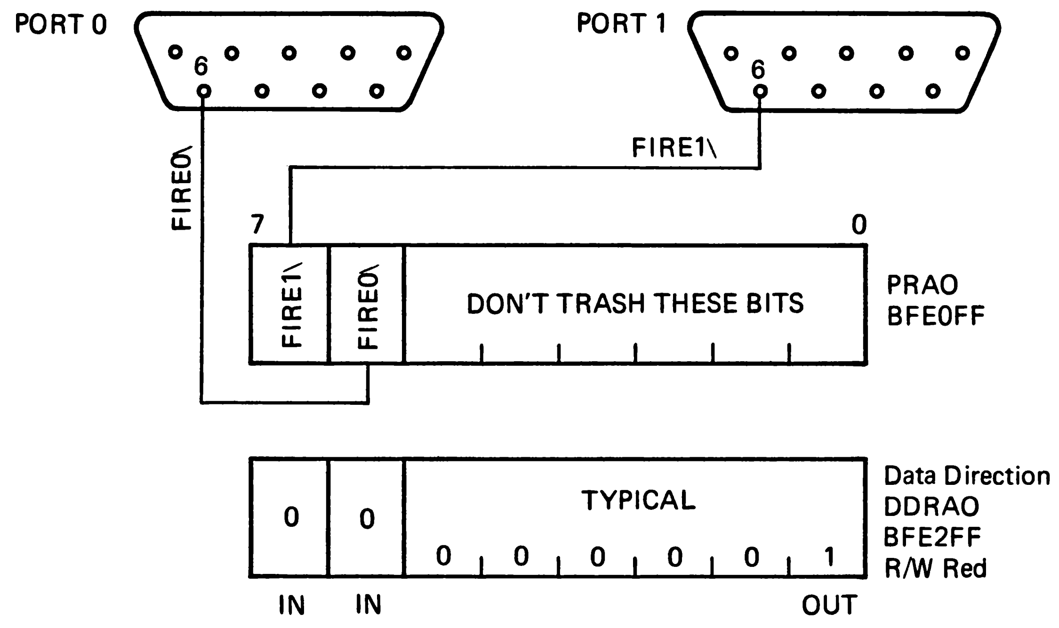
************** PART 3 - INTERNAL CONNECTORS *******************

DISK INTERNAL ...34 PIN RIBBON (J10)

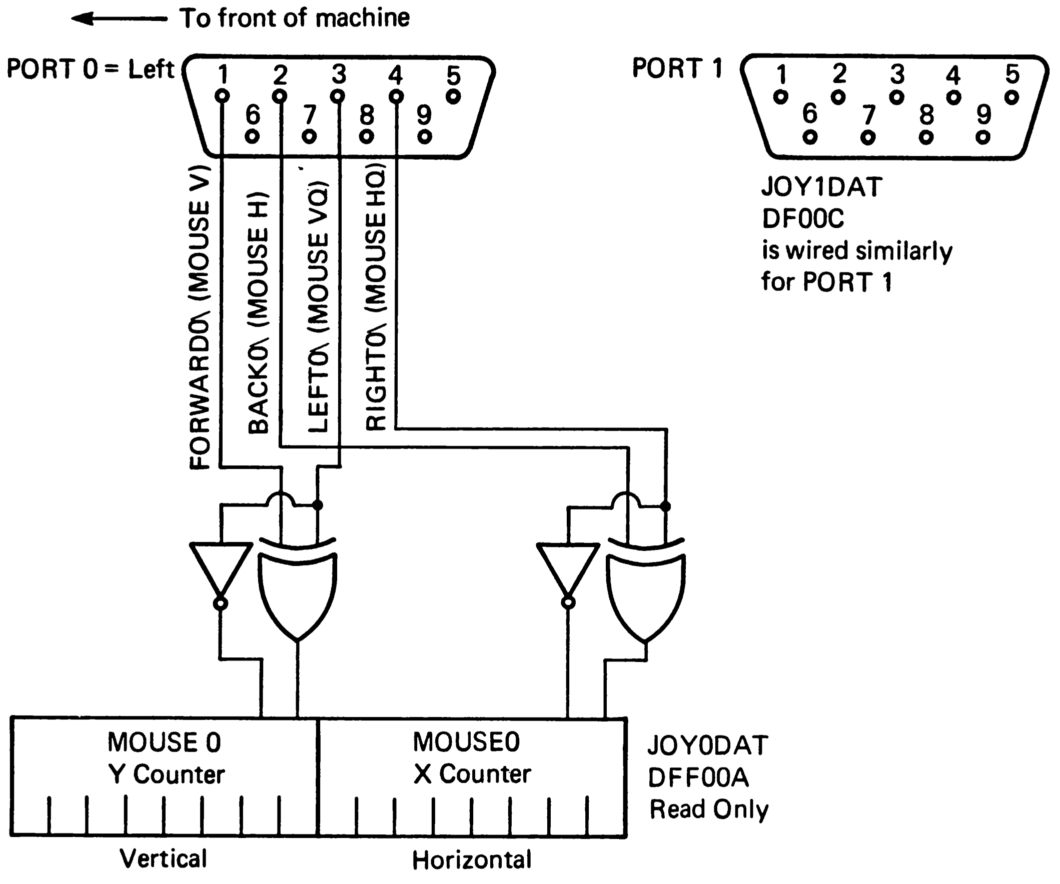
1	GND	18	DIRB
2	CHNG*	19	GND
3	GND	20	STEPB*
4	MTR0D*(led	21	GND
5	GND	22	DKWDB*
6	N.C	23	GND
7	GND	24	DKWEB*
8	INDEX*	25	GND
9	GND	26	TK0*
10	SEL0B*	27	GND
11	GND	28	WPRO*
12	N.C.	29	GND
13	GND	30	DKRD*
14	N.C.	31	GND
15	GND	32	SIDEB*
16	MTR0D*	33	GND
17	GND	34	RDY*

DISK INTERNAL POWER ...4 PIN STRAIGHT (J13)

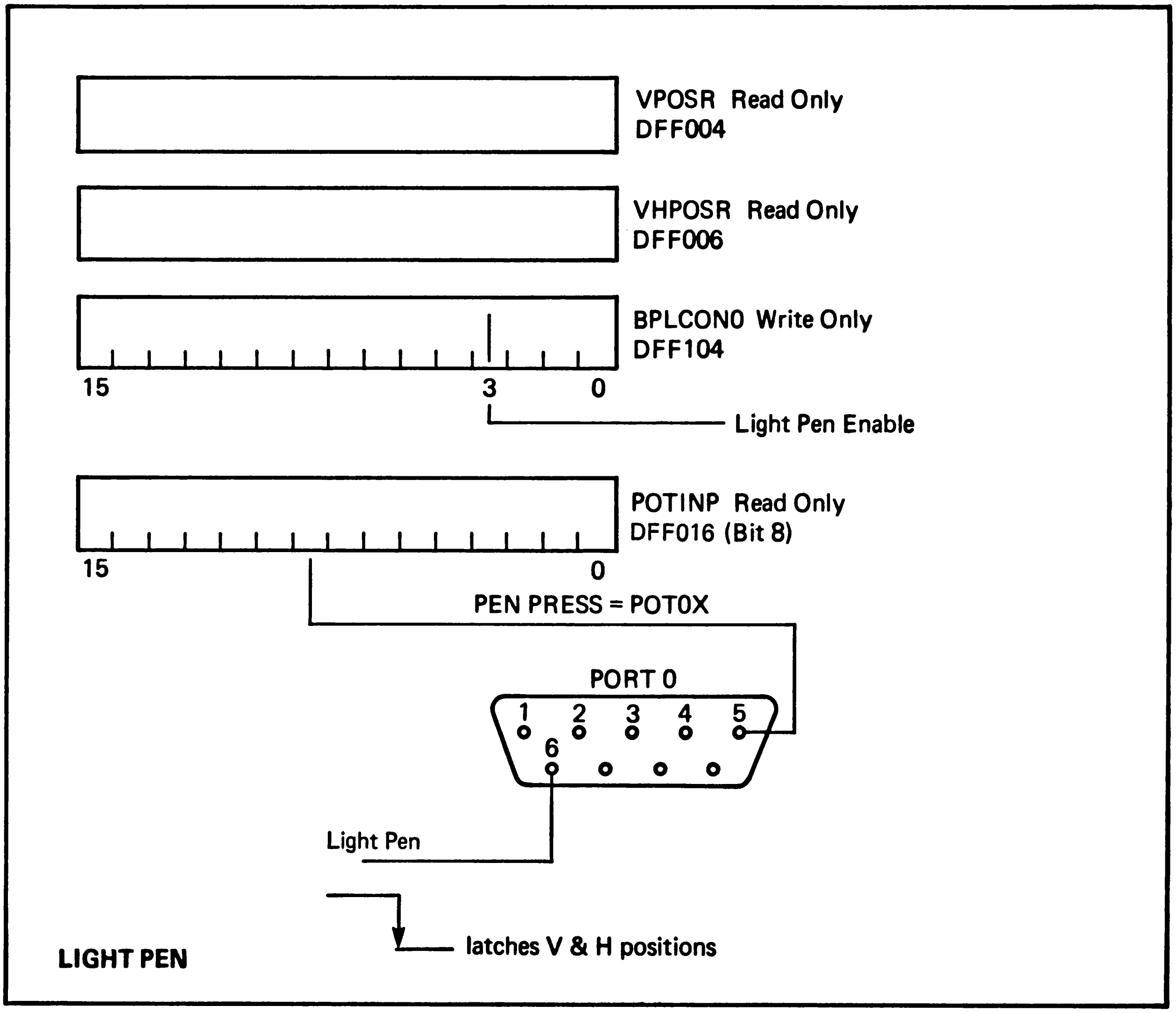
1	+12
2	GND
3	GND
4	+5


PORT 0
6
PORT 1
6
FIRE0\
FIRE1\
7
0
FIRE1\
FIRE0\
DON'T TRASH THESE BITS
PRAO
BFE0FF
0
0
TYPICAL
0 0 0 0 0 1
Data Direction
DDRAO
BFE2FF
R/W Red
IN
IN
OUT

Reading FIRE BUTTONS

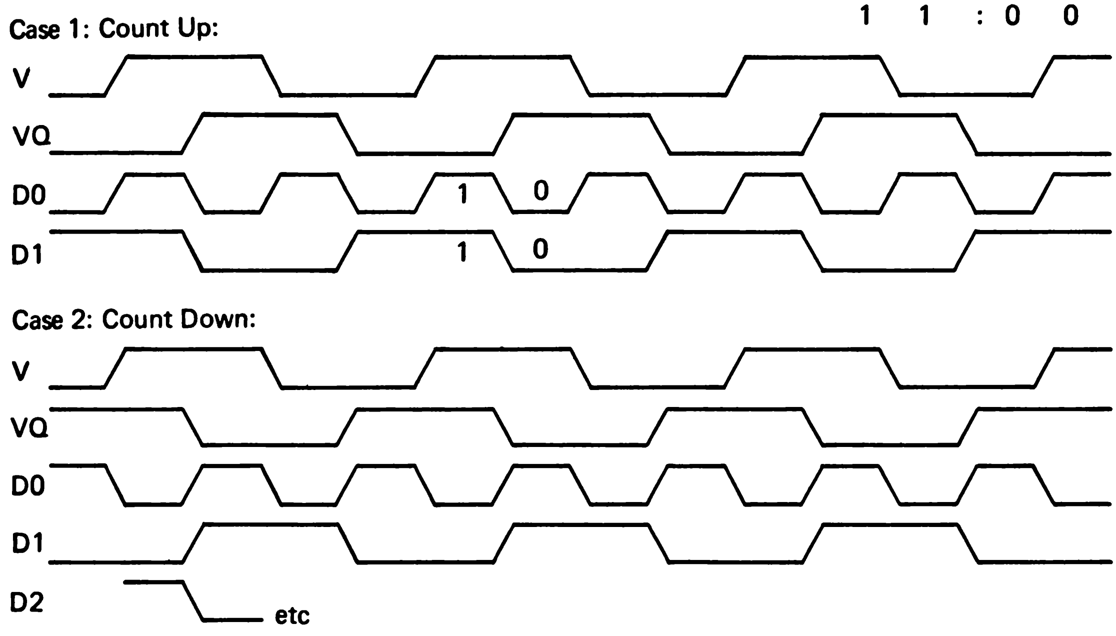

To front of machine
PORT 0 = Left
1 2 3 4 5
6 7 8 9
PORT 1
1 2 3 4 5
6 7 8 9
JOY1DAT
DF00C
is wired similarly
for PORT 1
FORWARD0\ (MOUSE V)
BACK0\ (MOUSE H)
LEFT0\ (MOUSE VQ)
RIGHT0\ (MOUSE HQ)
MOUSE 0
Y Counter
MOUSE0
X Counter
JOY0DAT
DFF00A
Read Only
Vertical
Horizontal

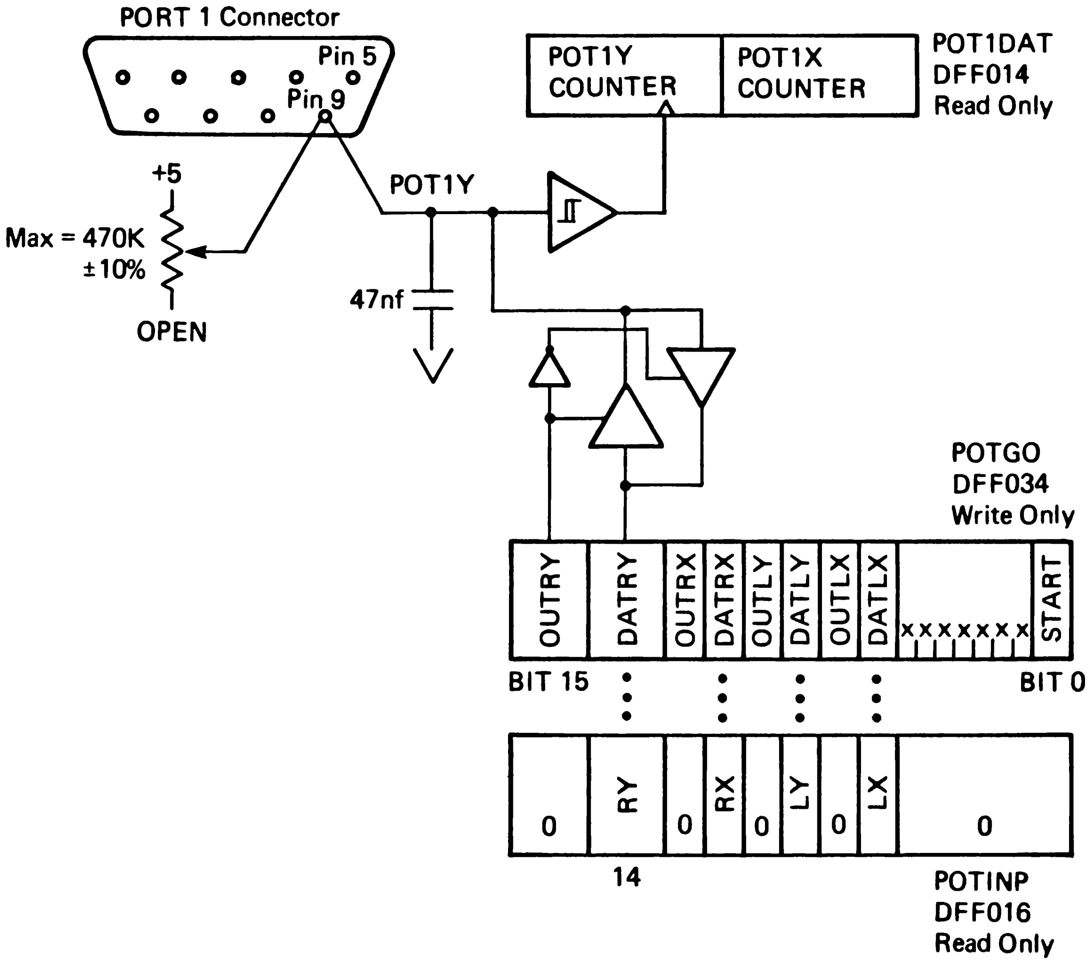
MOUSE Counters


VPOSR Read Only
DFF004
VHPOSR Read Only
DFF006
BPLCON0 Write Only
DFF104
15
3
0
Light Pen Enable
POTINP Read Only
DFF016 (Bit 8)
15
0
PEN PRESS = POT0X
PORT 0
1
2
3
4
5
6
Light Pen
latches V & H positions
LIGHT PEN

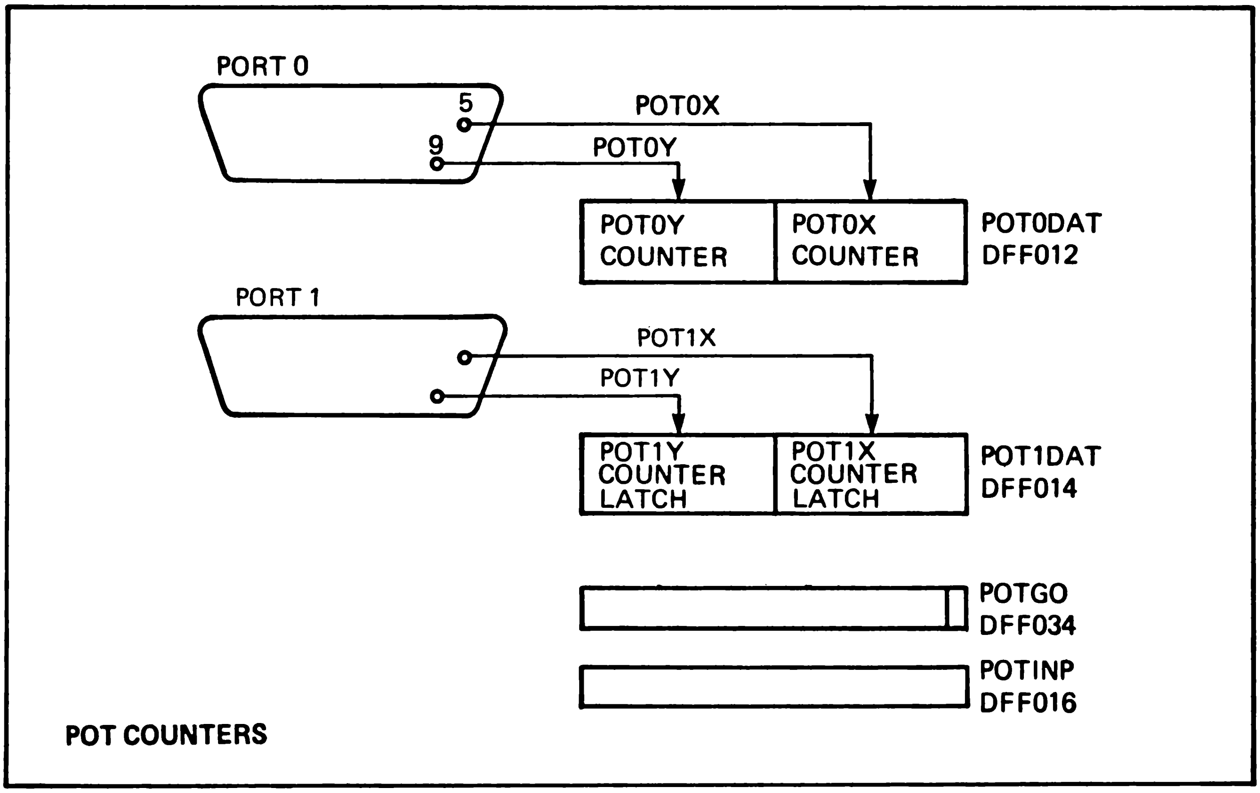
MOUSE QUADRATURE

V	VQ	:	D1	D0
0	0	:	1	0
0	1	:	0	1
1	0	:	1	1
1	1	:	0	0


Case 1: Count Up:
V
VQ
D0
1
0
D1
1
0
Case 2: Count Down:
V
VQ
D0
D1
D2
etc


PORT 1 Connector
Pin 5
Pin 9
+5
Max = 470K
±10%
OPEN
POT1Y
47nf
POT1Y COUNTER
POT1X COUNTER
POT1DAT
DFF014
Read Only
POTGO
DFF034
Write Only
OUTRY
DATRY
OUTRX
DATRX
OUTLY
DATLY
OUTLX
DATLX
XXXXXXX
START
BIT 15
BIT 0
0
RY
0
RX
0
LY
0
LX
0
14
POTINP
DFF016
Read Only

POT COUNTER


PORT 0
5
9
POT0X
POT0Y
POT0Y COUNTER
POT0X COUNTER
POT0DAT
DFF012
PORT 1
POT1X
POT1Y
POT1Y COUNTER LATCH
POT1X COUNTER LATCH
POT1DAT
DFF014
POTGO
DFF034
POTINP
DFF016

POT COUNTERS

Appendix F

Peripheral Interface Adapters

This appendix contains information about the 8520 peripheral interface adapters.

QUICK REFERENCE -- BRIEF ADDRESS MAP FOR 8520S

The system hardware selects the 8520s (also called CIAs) when the upper three address bits are 101. Furthermore, CIAA is selected when A12 is low, A13 high; CIAB is selected when A12 is high, A13 low.

You can use either byte or word addresses to access the 8520s. For byte access (seems to be the usual case), A0 must be 0 for CIAA, 1 for CIAB. For word access, CIAB communicates on data bits 15-8; CIAA communicates on data bits 7-0. (A0 is always 0 for word access, naturally.)

Address bits A11, A10, A9, and A8 are used to specify which of the 16 internal registers you want to access. This is indicated by "r" in the address. All other bits are don't cares. So, CIAA is selected by the following binary address: 101x xxxx xx01 rrrr xxxx xxx0. CIAB address: 101x xxxx xx10 rrrr xxxx xxx1

With future expansion in mind, we have decided on the following addresses: CIAA = BFEr01; CIAB = BFDr00.

CIAB Address Map

Byte Address	Register Name	Data bits 7	6	5	4	3	2	1	0
BFD000		/DTR	/RTS	/CD	/CTS	/DSR	SEL	POUT	BUSY
BFD100		/MTR	/SEL3	/SEL2	/SEL1	/SEL0	/SIDE	DIR	/STEP
BFD200		ddr for port A (BFD000); 1 = output (set to 0xC0)							
BFD300		ddr for port B (BFD100); 1 = output (set to 0xFF)							
BFD400		CIAB timer A low byte							
BFD500		CIAB timer A high byte							
BFD600		CIAB timer B low byte							
BFD700		CIAB timer B high byte							
BFD800		Horizontal sync event counter bits 7-0							
BFD900		Horizontal sync event counter bits 15-8							
BFDA00		Horizontal sync event counter bits 23-16							
BFDB00		not used							
BFDC00		CIAB serial data register							
BFDD00		CIAB interrupt control register							
BFDE00		CIAB Control register A							
BFDF00		CIAB Control register B							

Note: CIAB can generate INT6.

CIAA Address Map

Byte Address	Register Name	Data bits 7	6	5	4	3	2	1	0
BFE001		/FIR1	/FIR0	/RDY	/TK0	/WPRO	/CHNG	/LED	OVL
BFE101		Parallel port							
BFE201		ddr for port A (BFE001);1=output (set to 0x03)							
BFE301		ddr for port B (BFE101);1=output (can be in or out)							
BFE401		CIAA timer A low byte							
BFE501		CIAA timer A high byte							
BFE601		CIAA timer B low byte							
BFE701		CIAA timer B high byte							
BFE801		60 Hz event counter bits 7-0							
BFE901		60 Hz event counter bits 15-8							
BFEA01		60 Hz event counter bits 23-16							
BFEB01		not used							
BFEC01		CIAA serial data register (keyboard)							
BFED01		CIAA interrupt control register							
BFEE01		CIAA control register A							
BFEF01		CIAA control register B							

Note: CIAA can generate INT2.

****

INTERFACE SIGNALS

Clock input

The 02 clock is a TTL compatible input used for internal device operation and as a timing reference for communicating with the system data bus.

CS - chip-select input

The CS input controls the activity of the 8520. A low level on CS while 02 is high causes the device to respond to signals on the R/W and address (RS) lines. A high on CS prevents these lines from controlling the 8520. The CS line is normally activated (low) at 02 by the appropriate address combination.

R/W - read/write input

The R/W signal is normally supplied by the microprocessor and controls the direction of data transfers of the 8520. A high on R/W indicates a read (data transfer out of the 8520), while a low indicates a write (data transfer into the 8520).

RS3-RS0 - address inputs

The address inputs select the internal registers as described by the register map.

DB7-DB0 - data bus inputs/outputs

The eight data bus output pins transfer information between the 8520 and the system data bus. These pins are high impedance inputs unless CS is low and R/W and 02 are high, to read the device. During this read, the data bus output buffers are enabled, driving the data from the selected register onto the system data bus.

IRQ - interrupt request output

IRQ is an open drain output normally connected to the processor interrupt input. An external pull-up resistor holds the signal high, allowing multiple IRQ outputs to be connected together. The IRQ output is normally off (high impedance) and is activated low as indicated in the functional description.

RES - reset input

A low on the RES pin resets all internal registers. The port pins are set as inputs and port registers to zero (although a read of the ports will return all highs because of passive pull-ups). The timer control registers are set to zero and the timer latches to all ones. All other registers are reset to zero.

REGISTER MAP

Each 8520 has 16 registers that you may read or write. Here is the list of registers and the access address of each within the memory space dedicated to the 8520:

RS3	RS2	RS1	RS0	Register #(hex)	NAME	MEANING
0	0	0	0	0	PRA	Peripheral data register A
0	0	0	1	1	PRB	Peripheral data register B
0	0	1	0	2	DDRA	Data direction register A
0	0	1	1	3	DDRB	Direction register B
0	1	0	0	4	TALO	Timer A low register
0	1	0	1	5	TAHI	Timer A high register
0	1	1	0	6	TBLO	Timer B low register
0	1	1	1	7	TBHI	Timer B high register
1	0	0	0	8		Event LSB
1	0	0	1	9		Event 8-15
1	0	1	0	A		Event MSB
1	0	1	1	B		No connect
1	1	0	0	C	SDR	Serial data register
1	1	0	1	D	ICR	Interrupt control register
1	1	1	0	E	CRA	Control register A
1	1	1	1	F	CRB	Control register B

SOFTWARE NOTE:

The operating system kernel has already allocated the use of all four of the timers TA and TB in the 8520s. If you are running under control of the system exec, be aware of the following allocation of system resources:

8520A, timer A -- Commodore serial communications (if no serial communications is happening, timer becomes available).

8520A, timer B -- Video beam follower (used when synchronizing the blitter device to the video beam, see the description of QBSBlit() in the system software manual). If no beam-sync'ed blits are in process, this timer will be available.

8520B, timer A -- Keyboard (used continuously, whenever system Exec is in control).

8520B, timer B -- Virtual timer device (used continuously whenever system Exec is in control; used for task switching and interrupts).

REGISTER NAMES

The names of the registers within the 8520s are as follows. The address at which each is to be accessed is given in this list.

Address for:

8520-A	8520-B	NAME	EXPLANATION
			(write)/(read mode)
BFE001	BFD000	PRA	Peripheral data register A
BFE101	BFD100	PRB	Peripheral data register B
BFE201	BFD200	DDRB	Data direction register "A"
BFE301	BFD300	DDRA	Data direction register "B"
BFE401	BFD400	TALO	TIMER A low register
BFE501	BFD500	TAHI	TIMER A high register
BFE601	BFD600	TBLO	TIMER B low register
BFE701	BFD700	TBHI	TIMER B high register
BFE801	BFD800		Event LSB
BFE901	BFD900		Event 8 - 15
BFEA01	BFDA00		Event MSB
BFEB01	BFDB00		No connect
BFEC01	BFDC00	SDR	Serial data register
BFED01	BFDD00	ICR	Interrupt control register
BFEE01	BFDE00	CRA	Control register A
BFEF01	BFDF00	CRB	Control register B

REGISTER FUNCTIONAL DESCRIPTION:

I/O PORTS (PRA, PRB, DDRA, DDRB)

Ports A and B each consist of an 8-bit peripheral data register (PR) and an 8-bit data direction register (DDR). If a bit in the DDR is set to a 1, the corresponding bit position in the PR becomes an output. If a DDR bit is set to a 0, the corresponding PR bit is defined as an input.

When you READ a PR register, you read the actual current state of the I/O pins (PA0-PA7, PB0-PB7, regardless of whether you have set them to be inputs or outputs.

Ports A and B have passive pull-up devices as well as active pull-ups, providing both CMOS and TTL compatibility. Both ports have two TTL load drive capability.

In addition to their normal I/O operations, ports PB6 and PB7 also provide timer output functions.

HANDSHAKING

Handshaking occurs on data transfers using the PC output pin and the FLAG input pin. PC will go low on the third cycle after a port B access. This signal can be used to indicate "data ready" at port B or "data accepted" from port B. Handshaking on 16-bit data transfers (using both ports A and B) is possible by always reading or writing port A first. FLAG is a negative edge-sensitive input that can be used for receiving the PC output from another 8520 or as a general-purpose interrupt input. Any negative transition on FLAG will set the FLAG interrupt bit.

REG	NAME	D7	D6	D5	D4	D3	D2	D1	D0
0	PRA	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
1	PRB	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
2	DDRA	DPA7	DPA6	DPA5	DPA4	DPA3	DPA2	DPA1	DPA0
3	DDRB	DPB7	DPB6	DPB5	DPB4	DPB3	DPB2	DPB1	DPB0

INTERVAL TIMERS (TIMER A, TIMER B)

Each interval timer consists of a 16-bit read-only timer counter and a 16-bit write-only timer latch. Data written to the timer is latched into the timer latch, while data read from the timer is the present contents of the timer counter.

The latch is also called a prescalar in that it represents the countdown value which must be counted before the timer reaches an underflow (no more counts) condition. This latch (prescalar) value is a divider of the input clocking frequency. The timers can be used independently or linked for extended operations. Various timer operating modes allow generation of long time delays, variable width pulses, pulse trains, and variable frequency waveforms. Utilizing the CNT input, the timers can count external pulses or measure frequency, pulse width, and delay times of external signals.

Each timer has an associated control register, providing independent control over each of the following functions:

START/STOP

A control bit allows the timer to be started or stopped by the microprocessor at any time.

PB on/off

A control bit allows the timer output to appear on a port B output line (PB6 for timer A and PB7 for timer B). This function overrides the DDRB control bit and forces the appropriate PB line to become an output.

Toggle/pulse

A control bit selects the output applied to port B while the PB on/off bit is ON. On every timer underflow, the output can either toggle or generate a single positive pulse of one cycle duration. The toggle output is set high whenever the timer is started, and set low by RES.

One-shot/continuous

A control bit selects either timer mode. In one-shot mode, the timer will count down from the latched value to zero, generate an interrupt, reload the latched value, then stop. In continuous mode, the timer will count down from the latched value to zero, generate an interrupt, reload the latched value, and repeat the procedure continuously.

In one-shot mode, a write to timer-high (register 5 for timer A, register 7 for Timer B) will transfer the timer latch to the counter and initiate counting regardless of the start bit.

Force load

A strobe bit allows the timer latch to be loaded into the timer counter at any time, whether the timer is running or not.

INPUT MODES

Control bits allow selection of the clock used to decrement the timer. Timer A can count 02 clock pulses or external pulses applied to the CNT pin. Timer B can count 02 pulses, external CNT pulses, timer A underflow pulses, or timer A underflow pulses while the CNT pin is held high.

The timer latch is loaded into the timer on any timer underflow, on a force load, or following a write to the high byte of the pre-scalar while the timer is stopped. If the timer is running, a write to the high byte will load the timer latch but not the counter.

BIT NAMES on READ-register

REG	NAME	D7	D6	D5	D4	D3	D2	D1	D0
4	TALO	TAL7	TAL6	TAL5	TAL4	TAL3	TAL2	TAL1	TAL0
5	TAHI	TAH7	TAH6	TAH5	TAH4	TAH3	TAH2	TAH1	TAH0
6	TBLO	TBL7	TBL6	TBL5	TBL4	TBL3	TBL2	TBL1	TBL0
7	TBHI	TBH7	TBH6	TBH5	TBH4	TBH3	TBH2	TBH1	TBH0

BIT NAMES on WRITE-register

REG	NAME	D7	D6	D5	D4	D3	D2	D1	D0
4	TALO	PAL7	PAL6	PAL5	PAL4	PAL3	PAL2	PAL1	PAL0
5	TAHI	PAH7	PAH6	PAH5	PAH4	PAH3	PAH2	PAH1	PAH0
6	TBLO	PBL7	PBL6	PBL5	PBL4	PBL3	PBL2	PBL1	PBL0
7	TBHI	PBH7	PBH6	PBH5	PBH4	PBH3	PBH2	PBH1	PBH0

TIME OF DAY CLOCK

TOD consists of a 24-bit binary counter. Positive edge transitions on this pin cause the binary counter to increment. The TOD pin has a passive pull-up on it.

A programmable alarm is provided for generating an interrupt at a desired time. The alarm registers are located at the same addresses as the corresponding TOD registers. Access to the alarm is governed by a control register bit. The alarm is write-only; any read of a TOD address will read time regardless of the state of the ALARM access bit.

A specific sequence of events must be followed for proper setting and reading of TOD. TOD is automatically stopped whenever a write to the register occurs. The clock will not start again until after a write to the LSB event register. This assures that TOD will always start at the desired time.

Since a carry from one stage to the next can occur at any time with respect to a read operation, a latching function is included to keep all TOD information constant during a read sequence. All TOD registers latch on a read of MSB event and remain latched until after a read of LSB event. The TOD clock continues to count when the output registers are latched. If only one register is to be read, there is no carry problem and the register can be read "on the fly" provided that any read of MSB event is followed by a read of LSB Event to disable the latching.

BIT NAMES for WRITE TIME/ALARM or READ TIME

REG	NAME								
8	LSB Event	E7	E6	E5	E4	E3	E2	E1	E0
9	Event 8-15	E15	E14	E13	E12	E11	E10	E9	E8
A	MSB Event	E23	E22	E21	E20	E19	E18	E17	E16

WRITE
CRB7 = 0
CRB7 = 1 ALARM

SERIAL PORT (SDR)

The serial port is a buffered, 8-bit synchronous shift register. A control bit selects input or output mode.

INPUT MODE

In input mode, data on the SP pin is shifted into the shift register on the rising edge of the signal applied to the CNT pin. After eight CNT pulses, the data in the shift register is dumped into the serial data register and an interrupt is generated.

OUTPUT MODE

In the output mode, Timer A is used as the baud rate generator. Data is shifted out on the SP pin at 1/2 the underflow rate of Timer A. The maximum baud rate possible is 02 divided by 4, but the maximum usable baud rate will be determined by line loading and the speed at which the receiver responds to input data.

To begin transmission, you must first set up Timer A in continuous mode, and start the timer. Transmission will start following a write to the serial data register. The clock signal derived from Timer A appears as an output on the CNT pin. The data in the serial data register will be loaded into the shift register, then shifted out to the SP pin when a CNT pulse occurs. Data shifted out becomes valid on the next falling edge of CNT and remains valid until the next falling edge.

After eight CNT pulses, an interrupt is generated to indicate that more data can be sent. If the serial data register was reloaded with new information prior to this interrupt, the new data will automatically be loaded into the shift register and transmission will continue.

If no further data is to be transmitted after the eighth CNT pulse, CNT will return high and SP will remain at the level of the last data bit transmitted.

SDR data is shifted out MSB first. Serial input data should appear in this same format.

BIDIRECTIONAL FEATURE

The bidirectional capability of the serial port and CNT clock allows many 8520s to be connected to a common serial communications bus on which one 8520 acts as a master, sourcing data and shift clock, while all other 8520 chips act as slaves. Both CNT and SP outputs are open drain to allow such a common bus. Protocol for master/slave selection can be transmitted over the serial bus or via dedicated handshake lines.

REG	NAME	D7	D6	D5	D4	D3	D2	D1	D0
C	SDR	S7	S6	S5	S4	S3	S2	S1	S0

INTERRUPT CONTROL REGISTER (ICR)

There are five sources of interrupts on the 8520:

- -Underflow from Timer A (timer counts down past 0)
- -Underflow from Timer B
- -TOD alarm
- -Serial port full/empty
- -Flag

A single register provides masking and interrupt information. The interrupt control register consists of a write-only MASK register and a read-only DATA register. Any interrupt will set the corresponding bit in the DATA register. Any interrupt that is enabled by a 1-bit in that position in the MASK will set the IR bit (MSB) of the DATA register and bring the IRQ pin low. In a multichip system, the IR bit can be polled to detect which chip has generated an interrupt request.

When you read the DATA register, its contents are cleared (set to 0), and the IRQ line returns to a high state. Since it is cleared on a read, you must assure that your interrupt polling or interrupt service code can preserve and respond to all bits which may have been set in the DATA register at the time it was read. With proper preservation and response, it is easily possible to intermix polled and direct interrupt service methods.

You can set or clear one or more bits of the MASK register without affecting the current state of any of the other bits in the register. This is done by setting the appropriate state of the MSBit, which is called the set/clear bit. In bits 6-0, you yourself form a mask that specifies which of the bits you wish to affect. Then, using bit 7, you specify HOW the bits in corresponding positions in the mask are to be affected.

- o If bit 7 is a 1, then any bit 6-0 in your own mask word which is set to a 1 sets the corresponding bit in the MASK register. Any bit that you have set to a 0 causes the MASK register bit to remain in its current state.
- o If bit 7 is a 0, then any bit 6-0 in your own mask word which is set to a 1 clears the corresponding bit in the MASK register. Again, any 0 bit in your own mask word causes no change in the contents of the corresponding MASK register bit.

If an interrupt is to occur based on a particular condition, then that corresponding MASK bit must be a 1.

Example: Suppose you want to set the Timer A interrupt bit (enable the Timer A interrupt), but want to be sure that all other interrupts are cleared. Here is the sequence you can use:

```
movi.b      01111110B,A0
mov.b   A0,ICR     ;MSB is 0, means clear
                   ;any bit whose value is
                   ;1 in the rest of the byte

movi.b  10000001B,A0
mov.b   A0,ICR     ;MSB is 1, means set
                   ;any bit whose value is
                   ;1 in the rest of the byte
                   ;(do not change any values
                   ; wherein the written value
                   ; bit is a zero)
```

Read interrupt control register:

REG	NAME	D7	D6	D5	D4	D3	D2	D1	D0
D	ICR	IR	0	0	FLG	SP	ALRM	TB	TA

Write interrupt control MASK:

REG	NAME	D7	D6	D5	D4	D3	D2	D1	D0
D	ICR	S/C	x	x	FLG	SP	ALRM	TB	TA

CONTROL REGISTERS

There are two control registers in the 8520, CRA and CRB. CRA is associated with Timer A and CRB is associated with Timer B. The format of the registers is as follows:

CONTROL REGISTER A:

BIT	NAME	FUNCTION
0	START	1 = start Timer A, 0 = stop Timer A. This bit is automatically reset (= 0) when underflow occurs during one-shot mode.
1	PBON	1 = Timer A output on PB6, 0 = PB6 is normal operation.
2	OUTMODE	1 = toggle, 0 = pulse.
3	RUNMODE	1 = one-shot mode, 0 = continuous mode.
4	LOAD	1 = force load (this is a strobe input, there is no data storage; bit 4 will always read back a zero and writing a 0 has no effect.)
5	INMODE	1 = Timer A counts positive CNT transitions, 0 = Timer A counts 02 pulses.
6	SPMODE	1 = Serial port=output (CNT is the source of the shift clock) 0 = Serial port=input (external shift clock is required)

BIT MAP OF REGISTER CRA:

REG#	NAME	TOD IN	SPMODE	INMODE	LOAD	RUNMODE	OUTMODE	PBON	START
E	CRA	0=60Hz 1=50Hz	0=input 1=output	0=02 1=CNT	1=force load (strobe)	0=cont. 1=one-shot	0=pulse 1=toggle	0=PB6OFF 1=PB6ON	0=stop 1=start

|<-------- Timer A Variables ----------------->|

All unused register bits are unaffected by a write and forced to 0 on a read.

CONTROL REGISTER B:

BIT	NAME	FUNCTION
0	START	1 = start Timer B, 0 = stop Timer B. This bit is automatically reset (= 0) when underflow occurs during one-shot mode.
1	PBON	1 = Timer B output on PB7, 0 = PB7 is normal operation.
2	OUTMODE	1 = toggle, 0 = pulse.
3	RUNMODE	1 = one-shot mode, 0 = continuous mode.
4	LOAD	1 = force load (this is a strobe input, there is no data storage; bit 4 will always read back a zero and writing a 0 has no effect.)
6,5	INMODE	Bits CRB6 and CRB5 select one of four possible input modes for Timer B, as follows:

CRB6	CRB5	Mode Selected
0	0	Timer B counts 02 pulses
0	1	Timer B counts positive CNT transitions
1	0	Timer B counts Timer A underflow pulses
1	1	Timer B counts Timer A underflow pulses while CNT pin is held high.

BIT	NAME	FUNCTION
7	ALARM	1 = writing to TOD registers sets Alarm 0 = writing to TOD registers sets TOD clock. Reading TOD registers always reads TOD clock, regardless of the state of the Alarm bit.

BIT MAP OF REGISTER CRB:

REG #	NAME	ALARM	INMODE	LOAD	RUNMODE	OUTMODE	PBON	START
F	CRB	0=TOD 1=Alarm	00=02 01=CNT 10=Timer A 11=CNT+ Timer A	1=force load (strobe)	0=cont. 1=one-shot	0=pulse 1=toggle	0=PB7OFF 1=PB7ON	0=stop 1=start

|<-----------------Timer B Variables--------------->|

All unused register bits are unaffected by a write and forced to 0 on a read.

PORT SIGNAL ASSIGNMENTS

This part specifies how various signals relate to the available ports of the 8520. This information enables the programmer to relate the port addresses to the outside-world items (or internal control signals) which are to be affected. This part is primarily for the use of the systems programmer and should generally not be used by applications programmers. Systems software normally is configured to handle the setting of particular signals, no matter how the physical connections may change. In other words, if you have a version of the system software that matches the rev. level of the machine (normally a true condition), when you ask that a particular bit be set, you don't care which port that bit is connected to. Thus applications programmers should rely on system documentation rather than going directly to the ports. Note also that in this, a multi-tasking operating system, many different tasks may be competing for the use of the system resources. Applications programmers should follow the established rules for resource access in order to assure compatibility of their software with the system.

```
Address BFERFF  data bits 7-0  (A12*) (int2)

PA7..game port 1, pin 6 (fire button*)
PA6..game port 0, pin 6 (fire button*)
PA5..RDY*       disk ready*
PA4..TK0*       disk track 00*
PA3..WPRO*      write protect*
PA2..CHNG*      disk change*
PA1..LED*       led light (0=bright)
PA0..OVL        memory overlay bit
SP...KDAT       keyboard data
CNT..KCLK
PB7..P7         data 7
PB6..P6         data 6
PB5..P5         data 5     Centronics parallel interface
PB4..P4         data 4          data
PB3..P3         data 3
PB2..P2         data 2
PB1..P1         data 1
PB0..P0         data 0
PC...drdy*              centronics control
F....ack*
```

```
Address BFDRFE  data bits 15-8   (A13*) (int6)

PA7..com line DTR*, driven output
PA6..com line RTS*, driven output
PA5..com line carrier detect*
PA4..com line CTS*
PA3..com line DSR*
PA2..SEL        centronics control
PA1..POUT       paper out ---+
PA0..BUSY       busy    ---+ |
                           | |
SP...BUSY       commodore -+ |
CNT..POUT       commodore ---+

PB7..MTR*       motor
PB6..SEL3*      select external 3rd drive
PB5..SEL2*      select external 2nd drive
PB4..SEL1*      select external 1st drive
PB3..SEL0*      select internal drive
PB2..SIDE*      side select*
PB1..DIR        direction
PB0..STEP*      step*

PC...not used
F....INDEX*     disk index*
```

Appendix G

Amiga Auto-configuration Architecture

This appendix, which appeared in earlier versions of the *Amiga Hardware Reference Manual*, has been deleted. Also, appendix I, which was distributed as errata, should not be used.

For the latest information about the interface to the Amiga microprocessor bus, please contact the Technical Support Manager at Commodore Business Machines or Commodore-Amiga.

Appendix H

Keyboard

This appendix contains a description of the Amiga keyboard interface and the hardware of the Amiga keyboard.

```
********** PART 4 - PORT SIGNAL ASSIGNMENTS FOR 8520 ************

Address BFFR01  data bits 7-0  (A12*) (int2)

PA7..game port 1, pin 6 (fire button*)
PA6..game port 0, pin 6 (fire button*)
PA5..RDY*       disk ready*
PA4..TK0*       disk track 00*
PA3..WPRO*      write protect*
PA2..CHNG*      disk change*
PA1..LED*       led light (0=bright)
PA0..OVL        memory overlay bit

SP...KDAT       keyboard data
CNT..KCLK
PB7..P7         data 7
PB6..P6         data 6
PB5..P5         data 5     Centronics parallel interface
PB4..P4         data 4          data
PB3..P3         data 3
PB2..P2         data 2
PB1..P1         data 1
PB0..P0         data 0

PC...drdy*                 Centronics control
F....ack*


Address BFDRFE  data bits 15-8   (A13*) (int6)

PA7..com line DTR*, driven output
PA6..com line RTS*, driven output
PA5..com line carrier detect*
PA4..com line CTS*
PA3..com line DSR*
PA2..SEL        Centronics control
PA1..POUT       paper out ---+
PA0..BUSY       busy    ---+ |
                           | |
SP...BUSY       commodore -+ |
CNT..POUT       commodore ---+

PB7..MTR*       motor
PB6..SEL3*      select external 3rd drive
PB5..SEL2*      select external 2nd drive
PB4..SEL1*      select external 1st drive
PB3..SEL0*      select internal drive
PB2..SIDE*      side select*
PB1..DIR        direction
PB0..STEP*      step*

PC...not used
F....INDEX*     disk index*
```

Once sync is restored, the keyboard will have clocked a garbage character into the computer. That is why the key-up/key-down flag is always transmitted last. Since the keyboard clocks out 1's to restore sync, the garbage character thus transmitted will appear as a key release, which is less dangerous than a key hit.

Whenever the keyboard detects that it has lost sync, it will assume that the computer failed to receive the keycode that it had been trying to transmit. Since the computer is unable to detect lost sync, it is the keyboard's responsibility to inform the computer of the disaster. It does this by transmitting a "lost sync" code (value F9H = 11111001) to the computer. Then it retransmits the code that had been garbled.

Note: the only reason to transmit the "lost sync" code to the computer is to alert the software that something may be screwed up. The "lost sync" code does not help the recovery process, because the garbage keycode can't be deleted, and the correct key code could simply be retransmitted without telling the computer that there was an error in the previous one.

Power-up sequence:

There are two possible ways for the keyboard to be powered up under normal circumstances: the computer can be turned on with the keyboard plugged in, or the keyboard can be plugged into an already "on" computer. The keyboard and computer must handle either case without causing any upset.

The first thing the keyboard does on power-up is to perform a self-test. This involves a ROM checksum test, simple RAM test, and watchdog timer test. Whenever the keyboard is powered up (or restarted -- see below), it must not transmit anything until it has achieved synchronization with the computer. The way it does this is by slowly clocking out 1 bits, as described above, until it receives a handshake pulse.

If the keyboard is plugged in before power-up, the keyboard may continue this process for several minutes as the computer struggles to boot up and get running. The keyboard must continue clocking out 1s for however long is necessary, until it receives its handshake.

If the keyboard is plugged in after power-up, no more than eight clocks will be needed to achieve sync. In this case, however, the computer may be in any state imaginable but must not be adversely affected by the garbage character it will receive. Again, because it receives a key release, the damage should be minimal. The keyboard driver must anticipate this happening and handle it, as should any application that uses raw keycodes.

Note: the keyboard must not transmit a "lost sync" code after resyncing due to a power-up or restart; only after resyncing due to a handshake time-out.

Once the keyboard and computer are in sync, the keyboard must inform the computer of the results of the self-test. If the self-test failed for any reason, a "selftest failed" code (value FCH = 11111100) is transmitted (the keyboard does not wait for a handshake pulse after sending the "selftest failed" code). After this, the keyboard processor goes into a loop in which it blinks the CAPS LOCK LED to inform the user of the failure. The blinks are coded as bursts of one, two, three, or four blinks, approximately one burst per second. One blink = ROM checksum failure; two blinks = RAM test failed; three blinks = watchdog timer test failed; four blinks = a short exists between two row lines or one of the seven special keys (this last test isn't implemented yet).

If the self-test succeeds, then the keyboard will proceed to transmit any keys that are currently down. First, it sends an "initiate powerup key stream" code (value FDH = 11111101), followed by the key codes of all depressed keys (with keyup/down set to "down" for each key). After all keys are sent (usually there won't be any at all), a "terminate key stream" code (value FEH = 11111110) is sent. Finally, the CAPS LOCK LED is shut off. This marks the end of the start-up sequence, and normal processing commences.

Note: These special codes, (that is, FCH et al) are 8-bit numbers; there is no up/down flag associated with them. However, the transmission bit order is the same as previously described.

The usual sequence of events will therefore be: power up; synchronize; transmit "initiate powerup key stream" (FDH); transmit "terminate key stream" (FEH).

Hard Reset

The keyboard has the additional task of resetting the computer on the command of the user. The user initiates hard reset by simultaneously pressing the CTRL key and the two "AMIGA" keys. The keyboard responds to this input by pulling KCLK low and starting a 500-ms timer. At the end of the 500 ms, the processor checks the three keys to see if they are still down, and if so, restarts the 500-ms timer. This continues until one or more of the three keys is released.

When one or more keys is released, then the processor will wait until the end of the 500 ms. Then it jumps to its start-up code, which releases KCLK and restarts the keyboard.

Special Codes

The special codes that the keyboard uses to communicate with the main unit are summarized here.

Code	Name	Meaning
F9		Last key code bad, next code is the same code retransmitted (used when keyboard and main unit get out of sync).
FA		Keyboard output buffer overflow
FB		Unused
FC		Keyboard selftest failed
FD		Initiate power-up key stream
FE		Terminate key stream
FF		Unused

KEYBOARD HARDWARE

This is a description of the hardware insides of the Amiga keyboard.

This description is valid only for the second revision of the keyboard, the version with the watchdog timer.

PROCESSOR

The processor is a Rockwell/NCR/MOS Technologies 6500/1. It contains 2K bytes of ROM, 64 bytes of RAM, and 4 I/O ports of 8 bits each. It also has a 16-bit timer and edge detect capability on two of the I/O lines (port A bits 0 and 1). It has a built-in crystal oscillator, running at 3.00 megahertz, which is divided internally to a 1.5 MHz internal clock.

RESET CIRCUITRY

There is a circuit for resetting the processor on power-on. The reset pulse lasts about 1 second after power is applied. The circuit also performs a "watchdog" function: once the processor starts scanning the key matrix, the watchdog timer is armed and will reset the processor if the scanning stops for more than about 50 milliseconds. The column 15 line is the trigger for the watchdog timer.

KEY MATRIX

There are 91 keys on the keyboard. 84 of them are arranged in a matrix of 6 rows and 15 columns (leaving six holes in the matrix). Each row is an input and has a pullup resistor to VCC on it (R=3.3K to 11K). Each column is an open-collector output with no pullup, i.e., it can drive a column line low, but not high. The program will drive columns one at a time and read rows.

The other seven keys are special shift keys as follows: CTRL, left SHIFT, right SHIFT, left ALT, right ALT, left AMIGA, right AMIGA. Each of these keys has a dedicated input on the microprocessor. The actual port and bit numbers of all the keys are described below.

PORTS

As mentioned, there are four I/O ports of 8 bits each. The following table describes each port and the meaning of each bit:

PORT A -- 6500/1 address 080 hex

PA.0	In/Out	KDAT output/positive edge detect input (*)
PA.1	Out	KCLK output (*)
PA.2	In	Row 0 input (low = switch closed).
PA.3	In	Row 1 input
PA.4	In	Row 2 input
PA.5	In	Row 3 input
PA.6	In	Row 4 input
PA.7	In	Row 5 input

(*) These two bits are swapped from the previous code, to take advantage of the positive edge-detect capability of the PA.0 pin (it is easier to detect a handshake this way).

PORT B -- 6500/1 address 081 hex

PB.0	In	Right SHIFT key input (low = switch closed).
PB.1	In	Right ALT key input
PB.2	In	Right AMIGA key input
PB.3	In	CTRL key input
PB.4	In	Left SHIFT key input
PB.5	In	Left ALT key input
PB.6	In	Left AMIGA key input
PB.7	Out	CAPS LOCK LED control (high = LED on).

PORT C -- 6500/1 address 082 hex

PC.0	Out	Column 0 output (active low)
PC.1	Out	Column 1 output
PC.2	Out	Column 2 output
PC.3	Out	Column 3 output
PC.4	Out	Column 4 output
PC.5	Out	Column 5 output
PC.6	Out	Column 6 output
PC.7	Out	Column 7 output

PORT D -- 6500/1 address 083 hex

PD.0	Out	Column 8 output
PD.1	Out	Column 9 output
PD.2	Out	Column 10 output
PD.3	Out	Column 11 output
PD.4	Out	Column 12 output
PD.5	Out	Column 13 output
PD.6	Out	Column 14 output
PD.7	Out	Column 15 output (*)

(*) This keyboard has only 15 columns, numbered 0 to 14. However, the microprocessor software supports 16 columns, so we can use it in a future keyboard.

COUNTER PIN (input or output)

On the watchdog timer board, the counter pin is connected to the column 15 output. On the older non-watchdog version, the counter pin is unconnected. This provides the keyboard processor the ability to determine which type of board it it is installed in, so the new processor can work in old boards (with minor changes to the board).

NMI INPUT

This is connected to VCC and will therefore never turn on.

MATRIX TABLE

The following table shows which keys are readable in port A for each column you drive. The key code for each key is also included (in hex).

Column	Row 5 (Bit 7)	Row 4 (Bit 6)	Row 3 (Bit 5)	Row 2 (Bit 4)	Row 1 (Bit 3)	Row 0 (Bit 2)
15 (PD.7)	(spare) (0E)	(spare) (1C)	(spare) (2C)	(spare) (47)	(spare) (48)	(spare) (49)
14 (PD.6)	(spare) (5D)	(LEFT SHIFT) (30)	CAPS LOCK (62)	TAB (42)	~ ` (00)	ESC (45)
13 (PD.5)	(spare) (5E)	Z (31)	A (20)	Q (10)	! 1 (01)	(spare) (5A)
12 (PD.4)	9 (N.P.) (3E)	X (32)	S (21)	W (11)	@ 2 (02)	F1 (50)
11 (PD.3)	6 (N.P.) (2E)	C (33)	D (22)	E (12)	# 3 (03)	F2 (51)
10 (PD.2)	3 (N.P.) (1E)	V (34)	F (23)	R (13)	$ 4 (04)	F3 (52)
9 (PD.1)	. (N.P.) (3C)	B (35)	G (24)	T (14)	% 5 (05)	F4 (53)

Column	Row 5 (Bit 7)	Row 4 (Bit 6)	Row 3 (Bit 5)	Row 2 (Bit 4)	Row 1 (Bit 3)	Row 0 (Bit 2)
8 (PD.0)	8 (N.P.) (3E)	N (36)	H (25)	Y (15)	^ 6 (06)	F5 (54)
7 (PC.7)	5 (N.P.) (2E)	M (37)	J (26)	U (16)	& 7 (07)	(spare) (5B)
6 (PC.6)	2 (N.P.) (1E)	< , (38)	K (27)	I (17)	* 8 (08)	F6 (55)
5 (PC.5)	ENTER (N.P.) (43)	> . (39)	L (28)	O (18)	(9 (09)	(spare) (5C)
4 (PC.4)	7 (N.P.) (3D)	? / (3A)	: ; (29)	P (19)	) 0 (0A)	F7 (56)
3 (PC.3)	4 (N.P.) (2D)	(spare) (3B)	" ' (2A)	{ [(1A)	_ - (0B)	F8 (57)
2 (PC.2)	1 (N.P.) (1D)	SPACE BAR (40)	(RET) (2B)	}] (1B)	+ = (0C)	F9 (58)
1 (PC.1)	0 (N.P.) (0F)	BACK SPACE (41)	DEL (46)	RET (44)	| \\ (0D)	F10 (59)
0 (PC.0)	- (N.P.) (4A)	CURS DOWN (4D)	CURS RIGHT (4E)	CURS LEFT (4F)	CURS UP (4C)	HELP (5F)

The following table shows which keys are readable in port B (shift keys).

(Bit 6)	(Bit 5)	(Bit 4)	(Bit 3)	(Bit 2)	(Bit 1)	(Bit 0)
LEFT AMIGA (66)	LEFT ALT (64)	LEFT SHIFT (60)	CTRL (63)	RIGHT AMIGA (67)	RIGHT ALT (65)	RIGHT SHIFT (61)

Glossary

Aliasing distortion	A side effect of sound sampling, where two additional frequencies are produced, distorting the sound output.
Alt keys	Two keys on the keyboard to the left and right of the Amiga keys.
Amiga keys	Two keys on the keyboard to the left and right of the space bar.
AmigaDOS	The Amiga operating system.
Amplitude	The voltage or current output expressed as volume from a sound speaker.
Amplitude modulation	A means of increasing audio effects by using one audio channel to alter the amplitude of another.
Attach mode	In sprites, a mode in which a sprite uses two DMA channels for additional colors. In sound production, combining two audio channels for frequency/amplitude modulation or for stereo sound.
Automatic mode	In sprite display, the normal mode in which the sprite DMA channel, once it starts up, automatically retrieves and displays all of the data for a sprite. In audio, the normal mode in which the system retrieves sound data automatically through DMA.
Barrel shifter	Blitter circuit that allows movement of images on pixel boundaries.
Baud rate	Rate of data transmission through a serial port.
Beam counters	Registers that keep track of the position of the video beam.
Bit-map	The complete definition of a display in memory, consisting of one or more bit-planes and information about how to organize the rectangular display.

Bit-plane	A contiguous series of display memory words, treated as if it were a rectangular shape.
Bit-plane animation	A means of animating the display by moving around blocks of playfield data with the blitter.
Blanking interval	Time period when the video beam is outside the display area.
Blitter	DMA channel used for data copying and line drawing.
Clear	Giving a bit the value of 0.
CLI	*See* command line interface.
Clipping	When a portion of a sprite is outside the display window and thus is not visible.
Collision	A means of detecting when sprites, playfields, or playfield objects attempt to overlap in the same pixel position or attempt to cross some pre-defined boundary.
Color descriptor words	Pairs of words that define each line of a sprite.
Color indirection	The method used by Amiga for coloring individual pixels in which the binary number formed from all the bits that define a given pixel refers to one of the 32 color registers.
Color palette	*See* Color table.
Color register	One of 32 hardware registers containing colors that you can define.
Color table	The set of 32 color registers.
Command line interface	The command line interface to system commands and utilities.
Composite video	A video signal, transmitted over a single coaxial cable, which includes both picture and sync information.
Controller	Hardware device, such as mouse or light pen, used to move the pointer or furnish some other input to the system.
Coordinates	A pair of numbers shown in the form (x,y), where x is an offset from the left side of the display or display window and y is an offset from the top.

Copper	Display-synchronized coprocessor that resides on one of the Amiga custom chips and directs the graphics display.
Coprocessor	Processor that adds its instruction set to that of the main processor.
Cursor keys	Keys for moving something on the screen.
Data fetch	The number of words fetched for each line of the display.
Delay	In playfield horizontal scrolling, specifies how many pixels the picture will shift for each display field. Delay controls the speed of scrolling.
Depth	Number of bit-planes in a display.
Digital-to-analog converter	A device that converts a binary quantity to an analog level.
Direct memory access	An arrangement whereby intelligent devices can read or write memory directly, without having to interrupt the processor.
Display field	One complete scanning of the video beam from top to bottom of the video display screen.
Display mode	One of the basic types of display; for example, high or low resolution, interlaced or non-interlaced, single or dual playfield.
Display time	The amount of time to produce one display field, approximately 1/60th of a second.
Display window	The portion of the bit-map selected for display. Also, the actual size of the on-screen display.
DMA	*See* direct memory access.
Dual-playfield mode	A display mode that allows you to manage two separate display memories, giving you two separately controllable displays at the same time.
Equal-tempered scale	A musical scale where each note is the 12th root of 2 above the note below it.
Exec	Low-level primitives that support the AmigaDOS operating system.

Font	A set of letters, numbers, and symbols sharing the same size and design.
Frequency	The number of times per second a waveform repeats.
Frequency modulation	A means of changing sound quality by using one audio channel to affect the period of the waveform produced by another channel. Frequency modulation increases or decreases the pitch of the sound.
Genlock	An optional feature that allows you to bring in a graphics display from an external video source.
High resolution	A horizontal display mode in which 640 pixels are displayed across a horizontal line in a normal-sized display.
Hold-and-modify	A display mode that gives you extended color selection—up to 4,096 colors on the screen at one time.
Interlaced mode	A vertical display mode where 400 lines are displayed from top to bottom of the video display in a normal-size display.
Joystick	A controller device that freely rotates and swings from left to right, pivoting from the bottom of the shaft; used to position something on the screen.
Light pen	A controller device consisting of a stylus and tablet used for drawing something on the screen.
Low resolution	A horizontal display mode in which 320 pixels are displayed across a horizontal line in a normal-sized display.
Manual mode	Non-DMA output. In sprite display, a mode in which each line of a sprite is written in a separate operation. In audio output, a mode in which audio data words are written one at a time to the output.
Minterm	One of eight possible logical combinations of data bits from three different data sources.
Modulo	A number defining which data in memory belongs on each horizontal line of the display. Refers to the number of bytes in memory between the last word on one horizontal line and the beginning of the first word on the next line.

Mouse	A controller device that can be rolled around to move something on the screen; also has buttons to give other forms of input.
Multitasking	A system in which many tasks can be operating at the same time, with no task forced to be aware of any other task.
Non-interlaced mode	A display mode in which 200 lines are displayed from top to bottom of the video display in a normal-sized display.
NTSC	National Television Standards Committee specification for composite video.
Overscan	Area scanned by the video beam but not visible on the video display screen.
Paddle controller	A game controller that uses a potentiometer (variable resistor) to position objects on the screen.
PAL	A European television standard similar to (but incompatible with) NTSC. Stands for "Phase Alternate Line."
Parallel port	A connector on the back of the Amiga that is used to attach parallel printers and other parallel add-ons.
Pitch	The quality of a sound expressed as its highness or lowness.
Pixel	One of the small elements that makes up the video display. The smallest addressable element in the video display.
Playfield	One of the basic elements in Amiga graphics; the background for all the other display elements.
Playfield object	Subsection of a playfield that is used in playfield animation.
Playfield animation	*See* bit-plane animation.
Pointer register	Register that is continuously incremented to point to a series of memory locations.
Polarity	True or false state of a bit.
Potentiometer	An electrical analog device used to adjust some variable value.
Primitives	Amiga graphics, text, and animation library functions.

Quantization noise	Audio noise introduced by round-off errors when you are trying to reproduce a signal by approximation.
RAM	Random access (volatile) memory.
Raster	The area in memory that completely defines a bit-map display.
Read-only	Describes a register or memory area that can be read but not written.
Resolution	On a video display, the number of pixels that can be displayed in the horizontal and vertical directions.
ROM	*See* read-only memory.
Sample	One of the segments of the time axis of a waveform.
Sampling rate	The number of samples played per second.
Sampling period	The value that determines how many clock cycles it takes to play one data sample.
Scrolling	Moving a playfield smoothly in a vertical or horizontal direction.
Serial port	A connector on the back of the Amiga used to attach modems and other serial add-ons.
Set	Giving a bit the value of 1.
Shared memory	The RAM used in the Amiga for both display memory and executing programs.
Sprite	Easily movable graphics object that is produced by one of the eight sprite DMA channels and is independent of the playfield display.
Strobe address	An address you put out to the bus in order to cause some other action to take place; the actual data written or read is ignored.
Task	Operating system module or application program. Each task appears to have full control over its own virtual 68000 machine.

Timbre	Tone quality of a sound.
Trackball	A controller device that you spin with your hand to move something on the screen; may have buttons for other forms of input.
Transparent	A special color register definition that allows a background color to show through. Used in dual-playfield mode.
UART	The circuit that controls the serial link to peripheral devices, short for Universal Asynchronous Receiver/Transmitter.
Video priority	Defines which objects (playfields and sprites) are shown in the foreground and which objects are shown in the background. Higher-priority objects appear in front of lower-priority objects.
Video display	Everything that appears on the screen of a video monitor or television.
Write-only	Describes a register that can be written to but cannot be read.

Index

68000
  bus sharing, 189
  instead of Copper, 24
  interrupting, 24, 207
  normal cycle, 189
  synchronizing with the video beam, 205
  with special-purpose chips, 4
ADKCON
  in audio, 149, 151
  in disk control, 233-4
Aliasing
  audio, 154
Animation, 172
Area fill, 180-3
Attachment
  audio, 150
  sprites, 116
Audio
  aliasing distortion, 154-7
  channels
    attaching, 149, 161
    choosing, 136
  data, 137-8
  data length registers, 139
  data location registers, 137, 138
  data output rate, 140-43
  decibel values, 140, 160
  DMA, 137, 143, 147, 161
  equal-tempered scale, 158-9
  interrupts, 147, 210
  joining tones, 147-8
  low-pass filter, 155-7
  modulation
    amplitude, 149
    frequency, 149, 150 , 161
  noise reduction, 154
  non-DMA output, 157
  output jacks, 245
  period, 140-43
  period register, 143
  playing multiple tones, 149
  producing a steady tone, 145-6
  RF, 245
  sampling period, 141
  sampling rate, 141, 152, 156, 161
  state machine, 161-4
  stopping, 144
  system overhead, 153
  volume, 139-40, 160
  volume registers, 139
  waveform transitions, 152
AUDxLCH, 137
AUDxLCL, 137
AUDxLEN, 139
AUDxPER, 143
AUDxVOL, 139
Background color, 35
Barrel shifter, 177
Beam comparator, 121
Beam position
  comparison enable bits, 13
  detection of, 205-6
  in Copper use, 20
  registers, 206
  vertical, 12
Beam position counter, 205
Bit-planes
  coloring, 44-6
  DMA, 52
  in dual-playfield mode, 60
  setting the number of, 37

- setting the pointers, 43
- Blitter
  - address scanning, 171
  - addressing, 170
  - animation, 172
  - area filling
    - exclusive, 182-3
    - inclusive, 180-2
  - blitter-finished disable bit (BFD), 23
  - blitter-nasty bit, 191
  - block transfers, 177
  - common equations, 174
  - complete example, 193
  - copying, 167-8
  - DMA priority, 186
  - DMA time slots, 186
  - equation-to-minterm conversion, 175
  - interrupts, 211
  - LF control byte, 171-7
  - line drawing
    - octants, 184
    - registers, 184
  - line drawing mode, 183-5
  - logic equations, 172-5
  - logic operations, 171-7
  - masking, 178-9
  - minterms, 173-6
  - modulo, 168-70
  - modulo registers, 169
  - pointer registers, 168
  - sequence of bus cycles, 192
  - shifting, 177
  - Venn diagrams, 175-7
  - with the Copper, 23
  - zero detection, 179
- Blitter registers
  - in line-drawing mode, 183-4
- BLTCON0
  - in line drawing, 183
  - in logic operations, 171
  - in shift control, 178
  - in zero detection, 179
- BLTCON1
  - in area fill, 181, 182
  - in blitter addressing, 170
  - in line drawing, 183, 185
  - in shift control, 178
- BLTSIZE, 171
- BLTxMOD, 169
- BLTxPTH, 168
- BLTxPTL, 168
- BPL1MOD, 51
- BPL2MOD, 51
- BPLCON0
  - enabling color, 52
  - in dual-playfield mode, 64
  - in hold-and-modify mode, 80
  - in interlacing, 40
  - in resolution mode, 38
  - selecting bit-planes, 37
  - setting bits, 37
  - with light pen, 225
- BPLCON1
  - setting scrolling delay, 78
- BPLCON2
  - in dual-playfield priority, 64 , 200
- BPLxPTH, 43, 50, 67
- BPLxPTL, 43, 50, 67
- CLXCON, 203
- CLXDAT, 203
- Collision
  - control register, 203-4
  - detection register, 202-3
  - sprites-playfields, 202-4
- Color
  - attached sprites, 118
  - background color, 35
  - color indirection, 31
  - color table, 35
  - enabling, 52
  - in dual-playfield mode, 62
  - in hold-and-modify mode, 79
  - playfields, 31-3, 34-7, 44-5, 62-63, 86-90
  - sample register contents, 86
  - sprites, 96-8
- Color registers
  - contents, 36
  - loading, 36

  names of registers, 35
  sprites, 127-9
Color selection
  in high-resolution mode, 90
  in hold-and-modify mode, 89
  in low-resolution mode, 88
COLORx, 35
Comparator, 121
Controller port
  joystick, 217
  mouse, 217
  trackball, 217
Controllers
  joystick, 220
  light pen, 224-6
  mouse, 217-9
  potentiometers, 224
  proportional
    joystick, 220-3
    paddle, 220-3
    registers, 223
  special, 226-7
  types, 6
  typical connections, 222
COP1LCH, 13
COP1LCL, 13
COP2LCH, 13
COP2LCL, 13
COPCON, 15
COPJMP1, 14
COPJMP2, 14
Copper
  affecting registers, 14
  bus cycles used, 9
  comparison enable, 21
  control register, 14
  danger bit (CDANG), 15
  features, 8
  horizontal beam position, 12
  in interlaced mode, 22
  in memory operations, 9
  in vertical blanking interrupts, 210
  instruction lists, 15, 17
  instructions
    description, 9
    MOVE, 9
    ordering, 16
    SKIP, 20-1
    summary, 25
    WAIT, 11, 19, 21
  interrupt, 210
  interrupting the 68000, 24
  jump strobe addresses, 14
  location registers, 13, 19, 21
  loops and branches, 21
  MOVE instruction, 9
  SKIP instruction, 20-1
  starting, 14, 19
  stopping, 19
  vertical beam position, 12
  WAIT instruction, 11, 19, 21
  with the blitter, 23
Coprocessor
  (see Copper), 7
Copying data, 167-8
Data-fetch
  high-resolution, 51
  in basic playfield, 49-51
  in horizontal scrolling, 76
Data-fetch start
  normal, 49
Data-fetch stop
  normal, 49
DDFSTOP, 49, 72, 76
DDFSTRT, 49, 72, 76
Decibel values, 160
Disk
  8520 ports, 228
  control, 228-9
  control register, 233
  controller, 5, 227-235
  data buffer, 232
  data pointer registers, 230
  data transfer, 230
  DMA, 230
  DMA buffer, 235
  drives, 5
  input stream synchronization register,

    235
  interrupts, 211, 235
  selection, 228-9
  sensing, 228-9
  write, 230
Display
  size of, 46
Display field, 29
Display memory, 46
Display modes, 30
Display output connector, 246
Display window
  positioning, 46
  size
    maximum, 72
    normal, 47
  starting position
    horizontal, 47, 70
    vertical, 47, 70
  stopping position
    horizontal, 48, 71
    vertical, 48, 72
DIWSTOP, 48, 71
DIWSTRT, 47, 69
DMA
  audio, 143, 161
  bit-planes, 52
  blitter, 186-92
  control, 212-3
  control register, 212
  disk, 230-3
  playfield, 52
  sprites, 105, 123
DMACON
  BLTPRI bit, 191
  in audio, 143
  in blitter logic operations, 179
  in playfields, 52
  stopping the Copper, 20 , 212
DMACONR, 212
DSKBLK, 235
DSKBYTR, 232
DSKDAT, 235
DSKLEN, 230
DSKPTH, 230
DSKPTL, 230
DSKSYNC, 235
Dual playfields
  bit-plane assignment, 60
  description, 58
  enabling, 64
  high-resolution colors, 63
  in high-resolution mode, 63
  low-resolution colors, 62
  priority, 64
  scrolling, 64
Dual-playfield mode, 33
External interrupts, 209
Field time, 29
Genlock
  effect on background color, 36
  in playfields, 82
High resolution
  color selection, 38, 90
  memory requirements, 42
  with dual playfields, 63
Hold-and-modify mode, 79
Horizontal blanking interval, 12
INTENA, 208
INTENAR, 208
Interlaced mode
  Copper in, 22
  memory requirements, 42
  modulo, 51
  setting interlaced mode, 39
Interrupts
  audio, 210
  blitter, 211
  control registers, 208-11
  Copper, 210
  disk, 211, 235
  during vertical blanking, 210
  external, 209
  interrupt enable bit, 209
  interrupt lines, 207
  maskable, 207
  nonmaskable, 207
  serial port, 211

- setting and clearing bits, 209

INTREQ, 24, 208
INTREQR, 208
JOY0DAT
- with joystick, 220
- with mouse/trackball, 218

JOY1DAT
- with joystick, 220
- with mouse/trackball, 218

Joystick
- proportional, 221
- reading, 220

Keyboard
- 8520, 236
- clock, 236
- ghosting, 239
- keycodes, 236-7
- reading, 236-9

Light pen
- controller port, 224-6
- registers, 225

Line drawing, 183
Low resolution
- color selection, 88

Manual mode
- in sprites, 119

Memory
- adding, 6
- primary and secondary, 5

Memory allocation
- audio, 137
- formula for playfields, 69
- playfields, 42
- sprite data, 100

Minterms, 173-6
Modulation
- amplitude, 149
- frequency, 150

Modulo
- blitter, 168-70
- in basic playfield, 50
- in horizontal scrolling, 76
- in interlaced mode, 51

Monitors, 246
Mouse
- buttons, 219
- counter, 218-9
- port, 218
- reading, 218-9

Noise
- audio, 154

Overscan, 46
Paddle controller, 220
Parallel port, 240
Peripherals, 5, 6
Pixels
- definition, 29
- in sprites, 95

Playfields
- allocating memory, 41
- bit-plane pointers, 43
- collision, 202-4
- color of pixels, 31-3
- color register contents, 86
- color table, 35
- coloring the bit-planes, 34, 44-6
- colors in a single playfield, 35
- data fetch, 49-51, 72
- defining a scrolled playfield, 78
- defining display window, 46-8
- defining dual playfields, 65
- defining the basic playfield, 53-5
- display window size
  - maximum, 72
  - normal, 47
- displaying, 52
- dual-playfield mode, 58
- enabling DMA, 52
- forming, 33
- high-resolution
  - color selection, 90
  - example, 56
  - mode, 30
- hold-and-modify, 89
- hold-and-modify mode, 79-82
- interlaced example, 56
- low-resolution
  - colors, 88

    - mode, 30
  - memory required, 41, 69
  - modulo registers, 51
  - multiple-playfield display, 82
  - normal, 30
  - pointer registers, 57, 67
  - priority, 200
  - register summary, 83-5
  - scrolling
    - horizontal, 74-8
    - vertical, 73-4
  - selecting bit-planes, 37
  - setting resolution mode, 38
  - specifying modulo, 50-2, 66-8
  - specifying the data fetch, 67
  - with external video source, 82
  - with genlock, 82
  - with larger display memory, 66-8
- Playfield-sprite priority, 200
- Ports
  - controller, 216
  - disk, 228
  - parallel, 240
  - serial, 240-5
- POT0DAT, 223
- POT1DAT, 223
- POTGO, 222, 226-7
- POTGOR, 226-7
- Priority
  - dual playfields, 64
  - playfield-sprite, 200
  - priority control register, 200
  - sprites, 198
- Proportional controllers
  - reading, 222
- Resolution
  - setting, 38
- Sampling
  - period, 141
  - rate, 152
- Scrolling
  - data fetch, 76
  - delay, 78
  - horizontal, 74-8
  - in dual-playfield mode, 64
  - in high-resolution mode, 75
  - modulo, 76
  - vertical, 73-4
- SERDATR, 241-3
- Serial port
  - baud rate, 240
  - output register, 243
  - receive-data register, 241
  - shift register, 243
- SERPER, 241
- Sound generation, 132-5
- Sprites
  - address pointers, 106
  - arming and disarming, 120
  - attached
    - color registers, 129
    - colors, 118
    - control word, 116
    - Copper list, 119
    - data words, 117, 119
  - clipped, 95
  - collision, 109, 202-4
  - color, 96-8
  - color registers used, 98
  - comparator, 121, 123
  - control registers, 121, 123, 125-6
  - control words, 102
  - data registers, 123, 126
  - data structure, 99-104
  - data words, 102
  - designing, 98
  - displaying
    - example, 106-8
    - steps in, 104
  - DMA, 105, 110
  - end-of-data words, 104
  - forming, 92-104
  - manual mode, 119
  - memory requirements, 100
  - moving, 108-10
  - overlapped, 114
  - parallel-to-serial converters, 120
  - pixels in sprites, 95

- pointer registers
  - initializing, 105
  - resetting, 106 , 124
- position registers, 121, 123
- priorities, 198
- priority, 111, 114, 200
- reuse, 110-13, 111
- screen position
  - horizontal, 92-4, 102
  - vertical, 94
- shape, 95
- size, 95
- vertical position, 102

SPRxCTL, 102, 120-1, 123, 125
SPRxDATA, 120, 123
SPRxDATB, 120, 123
SPRxPOS, 102, 120-1, 123, 125
SPRxPTH, 105, 123-4
SPRxPTL, 105, 123-4
Text
- packed, 178

Trackball
- counter, 218
- port, 218

VHPOSR
- with beam counter, 206
- with light pen, 225

VHPOSW
- with beam counter, 206

Video
- beam position, 12
- camera input, 5
- external sources, 82
- laser disk input, 5
- monitors, 5
- output, 245
- VCR input, 5

Volume, 139-40
VPOSR
- in playfields, 57
- with beam counter, 206
- with light pen, 225

VPOSW
- with beam counter, 206

Waveforms
- audio, 132

> $24.95 FPT USA

Amiga™ Technical Reference Series

Amiga Hardware Reference Manual

The Amiga Computer is an exciting new high-performance microcomputer with superb graphics, sound, and multitasking capabilities. Its technologically advanced hardware, designed around the Motorola 68000 microprocessor, includes three sophisticated custom chips that control graphics, audio, and peripherals. The Amiga's unique system software is contained in 192K of read-only memory (ROM), providing programmers with unparalleled power, flexibility, and convenience in designing and creating programs.

The AMIGA HARDWARE REFERENCE MANUAL, written by the technical staff at Commodore-Amiga, Inc., is an in-depth and thorough description of the Amiga's hardware. It is both an introduction to the design of the machine and a reference to its architecture. It includes:

- an introductory tutorial on writing assembly language programs to directly control the Amiga's graphics and hardware
- descriptions of the Copper (coprocessor), playfields, sprites, and the Blitter, as well as audio, system control, and interface hardware
- eight appendices giving a concise summary of the entire register set and the uses of individual bits
- a glossary of key terms

For the serious programmer working in assembly language, C, or Pascal who wants to take full advantage of the Amiga's impressive capabilities, the AMIGA HARDWARE REFERENCE MANUAL is an essential reference.

Written by the technical staff at Commodore-Amiga, Inc., who designed the Amiga's hardware and system software, the AMIGA HARDWARE REFERENCE MANUAL is the definitive source of information on the internal design and architecture of this revolutionary microcomputer.

The other books in the ***Amiga Technical Reference Series*** are:

Amiga Intuition Reference Manual
Amiga ROM Kernel Reference Manual: Libraries and Devices
Amiga ROM Kernel Reference Manual: Exec

Cover design by Marshall Henrichs
Cover photograph by Jack Haeger

Addison-Wesley Publishing Company, Inc.

ISBN 0-201-11077-6

This was brought to you

from the archives of

http://retro-commodore.eu